Theories of translation correctness for concurrent programming languages

Mark Millington

Doctor of Philosophy

University of Edinburgh

1985

ABSTRACT

This thesis addresses the problems of defining and proving translation correctness for programming languages describing concurrent processes. Taking an operationally motivated approach two distinct theories are proposed, a bisimulation theory and a testing theory, and both are articulated on substantial examples.

The bisimulation approach is applied to an example translation from a variant of CSP to CCS by defining a syntax-directed context-sensitive translation from CSP to CCS and establishing the correctness criteria of the bisimulation approach.

The testing approach provides two possible correctness criteria; implementation and complete-implementation, of which the latter implies the former. A substantial example of each is given. In demonstrating complete-implementation the example translates handshake communication in a CSP-like language to shared variable communication in an artificial language manipulating a communication state. For implementation we consider the design and specification of a concurrent sorting machine in CCS for which the design does not completely implement the specification.

Acknowledgements

My first thanks must go to Matthew Hennessy without whose patient advice and encouragement this thesis would certainly have been very much poorer and probably never finished at all. I would also like to express my gratitude to Rocco de Nicola and Wei Li for many talks which have had a fundamental influence on this research.

Most of all I would like to acknowledge the tremendous support given so freely by my family at all the times I needed it most, not just over the duration of this research but over all the years I can remember.

Lastly my thanks go to John Jones and other members of the Edinburgh AI department for putting up with me while I slowly finished this work.

The work reported here was supported by an SERC studentship.

Declaration

This thesis was composed by myself under the guidance of my supervisor Matthew Hennessy.

Contents

0. Introduction

It is commonly agreed that the subject of computer science is a fine example of layers of abstraction, each of which is intended to be self-contained and meaningful independently of the others but most of which may be expressed in terms of other, in some sense lower, levels. Consider for example a hypothetical implementation of the programming language Prolog.

> Prolog is implemented in the programming language C.
>
> C is implemented in assembly language.
>
> Assembly language is implemented in machine-code.
>
> Machine-code is implemented in register-transfer systems [Gordon 81].
>
> Register-transfer systems are implemented in VLSI structures [Gordon 81,Mead and Conway 80].
>
> VLSI structures are implemented in physical semiconductor devices [Mead and Conway 80].

The aim of this thesis is to investigate rigorous methods by which such claims might be given accurate statement and certain proof.

0.1. The problem of translation correctness

This has been an oft ignored problem for three reasons at least:

1. It has been largely traditional to say "the translation defines the semantics of the source language", so the

translation is correct by definition and the correctness problem goes away since in this case the meaning of a source program is (by definition) the meaning of its translation. At least one reason for adopting this approach has been the difficulties encountered in giving even simple languages an independent semantics. The advantage of an independent semantics is that it can be abstractly stated without irrelevant reference to particular implementation details and a number of different translations (to possibly different languages) proved correct relative to it.

2. Any non-trivial translation involves a great deal of case analysis and detail which carries over dramatically into any correctness proof; it's basically hard work.

3. It is often unclear in what exact sense a source program and its target translation have an acceptably similar meaning. Consider for example the much vaunted 'sameness' of a transistor and a (perfect) switch. They are different at least in the sense that one is physical and the other notional, and one manipulates continuous voltages over continuous time while the other manipulates discrete values (ones and zeros) over discrete time instants. The sense in which they are the same is really determined by an asymmetry: the transistor is used in such a way that its behaviour is analogous to that of a switch, so the switch is the senior partner in this sameness and the transistor is considered not in all its glory but rather in a diminished fashion. The particular diminution in this instance is the insistence that the transistor be used in

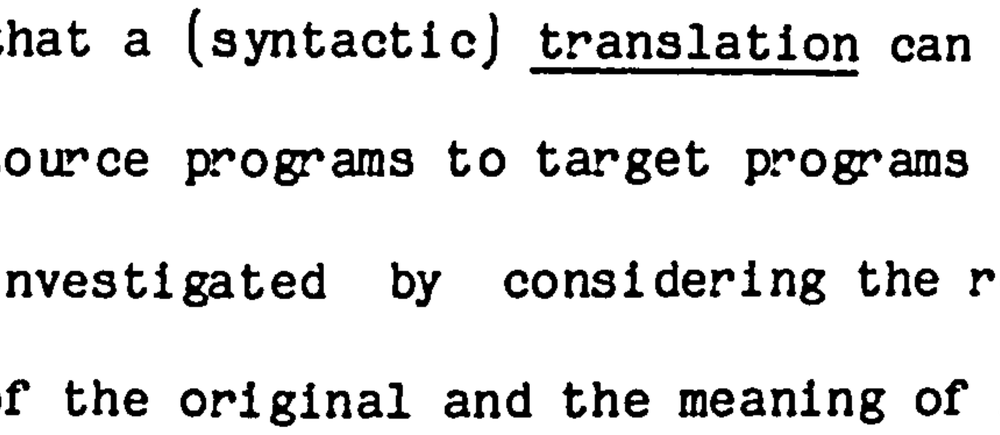
its saturation mode [Mead and Conway 80].

0.2. Some previous work in this area

Suppose there are two syntactic languages (sets of valid strings called programs) L_1 and L_2, the former of which is called the source and the latter the target . An exact formulation of the translation correctness problem will depend on how 'meaning' is ascribed to a program and what aspects of this meaning are to be preserved by a correct translation.

In the denotational approach to programming language semantics ([Gordon 79], [Stoy 77]) each language L is given an independent semantics by a semantic domain SD(L) of program 'meanings' (denotations) and a semantic mapping M_L which associates with each program of L a meaning in SD(L).

The approach of Morris [Morris 73] and the ADJ group [ADJ 79] is that a (syntactic) translation can be viewed as a mapping $[\![.]\!]$ from source programs to target programs and its correctness can be investigated by considering the relationship between the meaning of the original and the meaning of its translation. The statement of correctness for this approach is that the meaning of the translation is a (semantic) translation tr of the meaning of the original where tr is some function from $SD(L_1)$ to $SD(L_2)$. This can be represented as a commuting diagram



to express

$$\forall\ p \in L_1 .\ tr(M_{L_1}(p)) = M_{L_2}([\![p]\!])$$

This is nicely parameterised on tr, the exact relationship between the original and its correct translation depends on an acceptable tr. However, it is expressed in terms of denotational semantics whereas we are interested mainly in the semantics of concurrent programming languages for which there is (as yet) no adequate denotational approach.

In the operational approach to programming language semantics there are no meaning functions or semantic domains as such, but rather each syntactic program is associated with a semantic object, a configuration, by the choice of certain semantic components such as the initial state upon which the program will be executed.

The set C of all possible configurations forms one part of a labelled transition system, a quadruple <C,->,A,T>, which defines the possible actions in A by which one configuration can progress to another. This is achieved in the definition of the transition relation -> $\subseteq$ C x A x C for which we employ $c_1 \xrightarrow{a} c_2$, read as "configuration c_1 can perform the action a, thereby becoming the resulting configuration c_2", to denote $(c_1,a,c_2) \in$ ->. The set T $\subseteq$ C of terminal configurations is a set of configurations which are regarded as having terminated and can progress no further.

Thus an operational semantics describes how a program acts rather than associating it with some 'meaning' object and so necessitates a different approach to translation correctness where 'behaviours'

rather than 'meanings' are compared.

A common operational approach to translation is that any syntactic program translation induces a semantic translation between the transition systems associated with each language by its (independent) operational semantics. For example it may be that a configuration comprising a program with a particular state is semantically translated to a configuration comprising the syntactically translated program together with some function of the particular state. Further, there may be some translation from actions in the source system to actions in the target system.

The work of Astesiano and Zucca [Astesiano and Zucca 82] is an attempt to generalise previous research on comparing behaviours in an operational setting. A semantic translation tr is taken to be a function from configurations in the source transition system $\langle C_1, \rightarrow_1, A, T_1 \rangle$ to configurations in the target transition system $\langle C_2, \rightarrow_2, A, T_2 \rangle$. Note the actions are taken to be from the same set A and no use is made of the terminal sets.

The method of comparing behaviours is derived from the observational equivalence of [Milner 80] which may be loosely paraphrased as

> Two configurations are (behaviourally, observationally) <u>equivalent</u> if whenever either can perform a particular action so can the other in such a way that the resulting configurations are also equivalent.

For observational equivalence this loose paraphrase is formally stated by considering the relator F defined, for $R \subseteq C_1 \times C_2$ by

$\langle c_1, c_2 \rangle \varepsilon F(R)$ if

1. $c_1 \xrightarrow{\underline{a}}_1 c_1'$ implies $\exists\, c_2'.\ c_2 \xrightarrow{\underline{a}}_2 c_2',\ c_1'\ R\ c_2'$
2. $c_2 \xrightarrow{\underline{a}}_2 c_2'$ implies $\exists\, c_1'.\ c_1 \xrightarrow{\underline{a}}_1 c_1',\ c_1'\ R\ c_2'$

Then the observational equivalence of Milner, written here as $\simeq_M$, is defined by considering "equivalence up to n observations":

$\simeq_n = F^n(C_1 \times C_2),\ n \geq 0$ (where $F^0(C_1 \times C_2) = C_1 \times C_2$)

then defining

$c_1 \simeq_M c_2$ iff $\forall\, n \varepsilon$ Nat. $c_1 \simeq_n c_2$

A refinement introduced by Astesiano and Zucca was to recognise that some properties of configurations have no behavioural expression in the transition relation and thus cannot be discerned by the observational equivalence as stated. One particular property of concern is the ability of some agents to indulge in an infinite amount of internal work to the exclusion of all else. If a configuration $c \varepsilon C_i$ ($i \varepsilon \{1,2\}$) has this ability then we say it is divergent and write $div_i(c)$. Then, to distingish divergent agents from non-divergent (convergent) agents we can replace $C_1 \times C_2$ in the above definition by a subset which will not allow equivalence of convergent and divergent configurations:

$CONV = \{(c_1, c_2):\ div_1(c_1)$ iff $div_2(c_2)\} \subseteq C_1 \times C_2$

To achieve this (re)define

$F^0(C_1 \times C_2) = CONV$

and then

$c_1 \simeq_{AZ} c_2$ iff $\forall\, n \geq 0.\ c_1 \simeq_n c_2$

so that

$\simeq_{AZ} \subseteq CONV$

since

$$\forall\, n \geq 0.\ \simeq_{n+1} \subseteq \simeq_n$$

The correctness statement for Astesiano and Zucca can then be formally stated as:

$$tr \text{ is correct } \text{iff } \forall\, c_1 \in C_1.\ c_1 \simeq_{AZ} tr(c_1)$$

The main thrust of their work was to use translations as a means of providing a semantics for the source language, thus they had none of the problems of reconciling a semantics for the translated source object with an (independent) semantics for that source object. If we now consider our transistor example we find a number of weaknesses to this approach.

1. The actions of a switch are expressed in terms of zeros and ones while the actions of a transistor are commonly expressed in terms of continuously varying voltage waveforms. It might be very difficult to find a bijection between the two at all, let alone one for which the statement of correctness holds.

2. The switch is given no seniority in this approach, if it can be proved that a transistor implements a switch then it can also be proved that a switch implements a transistor.

3. It is not at all obvious that observational equivalence is the appropriate vehicle for a translation theory; under certain circumstances it can be too restrictive [de Nicola and Hennessy 84] or too liberal (in which case certain other requirements such as equality of final states in the configurations may need to be given an artificially behavioural aspect [Astesiano and Zucca 82]).

The operational approach is also adopted by Li [Li 83]; for Li a semantic translation is a function tr between configurations in labelled transition systems together with an injective function f from actions of the source to actions of the target. Two definitions of translation correctness are given, correctness and adequacy of which the latter implies the former and is employed mainly as a proof technique. The aims of Li's work are much closer to those of this thesis than Astesiano and Zuccas', being concerned with translation correctness for languages with independent sematics.

The essential approach of correctness is that the execution of a program can only be represented by a finite sequence of transitions ending in a terminated configuration (a successful computation), by a finite transition sequence ending in a non-terminated configuration (a deadlocked computation which is not terminated but cannot proceed) or by an infinite transition sequence. Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the source system and in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program.

Where the concept of correctness focusses on the result (especially the input-output relation) of a program and its translation, the concept of adequacy pays more attention to the detailed simulation of the behaviour of a program by its translation. Adequacy is stated in six clauses P0 to P5 [Li 83

page 212]; loosely P0 says that terminated configurations must be translated only to terminated configurations and conversely. P1 ensures that any transition of the source system can be simulated in the target system, P2 and P4 imply a weak version of the converse of P1, and P1 and P3 cover the simulation of infinite transition sequences. Finally, clause P5 implies that deadlocked configurations in the transition systems correspond with each other.

A number of inadequacies come to light if we consider applying these approaches to our transistor example, firstly correctness;

1. Transistors and switches don't have very interesting input-output relations, it's how they behave that is important.

2. The seniority of the switch has been overlooked, for example transistors can burn out if used improperly; is there any corresponding switch state?

Similar difficulties arise in the application of adequacy,

1. The requirements of P2 and P4 disregard the superior role of the switch over the transistor; the switch is required to act like a transistor.

2. It is not obvious that an injective function f can be found between actions of the source and actions of the target, we might expect many waveforms to correspond to (say) a one.

Another approach to the general translation problem as stated is to merge L_1 and L_2 into a language L sufficiently descriptive to span all the levels we are interested in. Languages in which this

approach have been taken contain the family of languages centred on CCS [Milner 80], including Circal [Milne 82], SCCS [Milner 82] and Real Time Agents [Cardelli 82], and the work by Gordon [Gordon 81] on register transfer systems. In such a case the problem is much reduced since there is typically a well-defined general (to L) sense in which two programs in L are "the same", for example they may have the same denotation or have equivalent behaviours.

0.3. The approaches presented here

This thesis contains two new theories of translation correctness within the framework of operational semantics. The first is a bisimulation theory of translation correctness in which the approach is very similar to that of Astesiano and Zucca but for two points:

1. The equivalence from which it is derived is the bisimulation equivalence of Park [Park 81b] rather than the observational equivalence of Milner. These two equivalences are very similar, indeed the earlier (loose) paraphrase of observational equivalence stands equally well for bisimulation equivalence. For bisimulation equivalence the paraphrase is again formally stated by considering the relator F defined earlier; the difference now is that the bisimulation equivalence, written here as $\sim_B$, is taken to be the maximal fixed-point of F, that is

$$\sim_B = F(\sim_B),$$

$$\forall\ R \subseteq C_1 \times C_2.\ (R \subseteq F(R) \text{ implies } R \subseteq \sim_B)$$

In fact

$$\simeq_B = \sum \{R: R \subseteq C_1 \times C_2,\ R \subseteq F(R)\}.$$

so a simple proof technique for $\simeq_B$ is thus

$$c_1 \simeq_B c_2 \quad \text{iff} \quad \exists R.\ c_1\ R\ c_2,\ R \subseteq F(R).$$

It can be seen since

$$\forall n \geq 0.\ \simeq_B \subseteq \simeq_n$$

that

$$\simeq_B \subseteq \simeq_M$$

and under a natural assumption that

$$\forall i \in \{1,2\}.\forall c \in C_i.\ \{c': \exists a.\ c \xrightarrow{a}_i c'\} \text{ is finite}$$

it can be shown [Hennessy and Milner 85] that

$$\simeq_B = \simeq_M$$

2. Following Astesiano and Zucca we would like to replace $C_1 \times C_2$ in the above definition by some subset of $C_1 \times C_2$. However, our approach is that the requirement of convergence is removed and replaced by a general parameter $P \subseteq C_1 \times C_2$ which is a relation between source configurations and target configurations relative to which correctness is to be established. The choice of P is part of the statement of correctness for any particular translation, allowing non-behavioural properties of configurations to be simply compared. The advantage of this approach over Astesiano and Zucca is that it is not necessary to distort the translation in an attempt to give certain properties (such as the state component of a terminated configuration) a behavioural expression (by communicating the state contents in a sequence of actions), the required constraints on acceptable correspondence can be expressed in P.

The statement of correctness for a semantic translation, a function tr from source configurations to target configurations, is given relative to a choice of $P \underline{C} C_1 \times C_2$ by

tr is P-good iff $\exists R \underline{C} P.\ R \underline{C} F(R), \forall c \in C.\ c\ R\ tr(c)$

In the second theory we adapt the testing equivalence of [de Nicola and Hennessy 84] to a <u>testing theory</u> of translation correctness. The basic idea is to translate not only objects but also how they are to be used (or, equivalently, how they can be tested). Thus for our switch and transistor example the object switch would be translated to the object transistor, and the uses of a switch described in terms of switch positions, zero's and one's will be translated to (say) gate voltages and continuous waveforms for use with the transistor.

In general then the source system will consist of source objects each paired with a prospective use, and their translations will consist of translated source objects paired with translated prospective uses.

A translation is now correct if it is a <u>complete implementation</u> , that is

1. A (source) object <u>mustnot</u> be amenable to a (source) use

iff

the translated object <u>mustnot</u> be amenable to the translated use

and

2. A (source) object <u>must</u> be amenable to a (source) use

iff

the translated object <u>must</u> be amenable to the translated use.

This approach respects the seniority of the switch since it only demands the transistor behave 'like' a switch when it is being used 'like' a switch.

All this is formalised by introducing the notion of an experiment system <E, ->,S> akin to a labelled transition system wherein the configurations e ε E are called experiments (being applications of tests to objects) and there are no action labels on the transition relation -> between experiments since experiments are assumed to progress by the communion of test and object with no external interference or communication. Further, some experiments e ε S $\underline{C}$ E are considered to be successful in the sense that the test has been satisfied.

The next step is to extend the notion of success to computations of experiments. A computation c ε comp(e) of an experiment e is a maximal (i.e. if it is finite then it cannot be extended) sequence of transition steps:

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots$$

Successful computations contain some e_i ε S, i.e. at some finite point they reach a successful experiment configuration. We can define the result of computation c by:

$$result(c) = \begin{cases} \top & \text{if c is successful} \\ \bot & \text{otherwise} \end{cases}$$

and then

$$resset(e) = \sum_{c \varepsilon comp[e]} result(c)$$

Now

e must (succeed) iff resset(e) = $\{\top\}$,

e mustnot (succeed) iff resset(e) = $\{\bot\}$.

An implementation preserves just these guarantees, so tr is an implementation if $\forall$ e ε E_1.

e <u>must</u> implies tr(e) <u>must</u>,

e <u>mustnot</u> implies tr(e) <u>mustnot</u>

In some cases we might like a stronger statement of complete implementation in which the above implications are replaced by biconditionals.

0.4. A chapter summary

In chapter 1 the basic concepts and notations concerning labelled transition systems are introduced and an example semantics is given for the concurrent programming language CCS [Milner 80]. Through this example it is shown how a structural operational semantics can be given using labelled transition systems.

Chapter 2 introduces a bisimulation theory of translation correctness in which the standard notion of bisimulation is extended and refined to a theory of translation correctness.

In chapter 3 the bisimulation theory is articulated on an example syntax-directed translation from CSP [Li 83] to CCS; a complete syntactic and semantic description is given of CSP and the translation is proved correct in the sense of the bisimulation theory. The translation employed is essentially that of Li [Li 83 chapter 6] except that certain restrictions have been lifted through the introduction of a context-sensitive aspect to the translation.

Chapter 4 motivates and presents a testing theory of translation correctness from which two (compatible) correctness criteria emerge; implementation and complete-implementation, of which the latter implies the former.

In chapter 5 complete-implementation is established for an example translation from a simple CSP-like language employing handshake style communication to a language in which communication takes place by altering shared variables. Both languages are given semantics, the translation is proved to be a complete-implementation, and it is shown why this example would present severe difficulties to the other approaches mentioned here.

Chapter 6 demonstrates how the notion of implementation introduced in chapter 4 can be employed within CCS to facilitate the task of specification. The example chosen is a machine to sort a multiset of numbers in a highly concurrent fashion; it is shown how certain aspects of the machines behaviour may be abstracted in its specification if the notion of implementation is employed rather than the more usual equivalence approach.

1. Labelled transition systems and operational semantics

This chapter is intended to introduce the concepts and techniques employed in the structural operational approach to programming language semantics [Plotkin 81], for use throughout the remainder of this thesis. The first section motivates the general approach and then goes on to introduce labelled transition systems as a formal representation for the approach. The second section presents, for later use, an example semantics for CCS.

1.1. Structural Operational Semantics

For sequential languages there have been four major approaches to programming language semantics: denotational semantics ([Gordon 79], [Stoy 77]), algebraic semantics [Guessarian 81], axiomatic semantics ([Hoare 69], [Dijkstra 76]), and operational semantics ([Landin 63],[Plotkin 81]). The most successful attempts, thus far, at providing a semantics for languages with concurrent and non-deterministic features have followed the operational approach, and in particular the structural operational approach. To introduce this method we can do little better than reproduce, with minor changes, the introduction given by Li in [Li 83 Introduction].

1.1.1. The general approach

Roughly speaking, the operational approach is to formally describe the execution of programs, i.e. to formalise the "operational nature" of programs. In general this purpose is achieved by specifying some convenient abstract machine and modelling the

execution of programs on that machine. This can indicate a way in which the language may be implemented.

Fortunately an operational semantics differs from many other approaches in that it does not require a lot of heavy mathematical machinery and is easy to understand.

A weakness of operational semantics is that because the semantics is based on an abstract machine it usually specifies some irrelevant details. This tends to make the semantics of any non-trivial language very obscure and detailed from the mathematical point of view. To overcome this weakness, or at least reduce it to a minimum, in this thesis we follow the structural operational approach or axiomatic operational approach developed by Plotkin and his colleagues. The basic ideas of this approach are:

1. To abstract away from the irrelevant details of the abstract machines we adopt some of the successful features of the denotational approach such as the use of abstract syntax to replace concrete syntax, and the viewing of states (stores) and environments as functions. Thus a simple configuration of an abstract machine can be written as

$$\langle S, s \rangle$$

where S denotes either the current statement to be executed or some token signifying that the program has terminated, for example

$$\langle done, s \rangle, \qquad \langle abort, s \rangle$$

may denote (respectively) a normal termination resulting in the state s and an abnormal termination resulting in the state s. We use C to denote the set of all configurations used within this

introduction.

To distinguish the successful executions from other computations (deadlocked and infinite computations) we introduce a set T of terminal configurations which is a subset of C. For example we can take

$$T = \{\langle done,s\rangle, \langle abort,s\rangle : s \text{ is a state}\}$$

2. We use labelled transition relations to model computation; thus a transition

$$p \xrightarrow{a} q$$

models one elementary execution step. This transition may be read "the configuration p may perform the action a, thereby becoming the configuration q". Here the action a denotes an internal action or interactive communication with some super system or the outside world. Let A be the set of possible transition actions in this introduction, then the labelled transition relation

$$\rightarrow \underline{C}\ C \times A \times C$$

describes the possible executions of programs. Execution of a program can be viewed as a transition sequence:

$$p_1 \xrightarrow{a1} p_2 \xrightarrow{a2} p_3 \xrightarrow{a3} \ldots$$

which is either infinite or finite. The crux of the matter lies in how to define the labelled transition relation which describes the semantics of a language. Let us consider how we could deal with two typical sequential programming language constructs in this approach:

The semantics of an assignment statement may be defined by the following axiom :

$$\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle done,s[v/x]\rangle$$

where v is the value taken by the expression e in the state s, and s[v/x] is the state which is identical with s except that it associates the value v with variable x. This transition can be interpreted as saying that the execution of the statement 'x:=e' in the state s results in a new configuration where the program has successfully terminated and the new state is the same as before except at x where it takes the value of e. The transition action ε should be interpreted to mean that the execution is performed without interaction with the outside world.

The semantics of a compound statement $S_1;S_2$ may be defined by the following rules:

1. If $\langle S_1,s\rangle \xrightarrow{a} \langle S_1',s'\rangle$

   then $\langle S_1;S_2,s\rangle \xrightarrow{a} \langle S_1';S_2,s'\rangle$

2. If $\langle S_1,s\rangle \xrightarrow{a} \langle done,s'\rangle$

   then $\langle S_1;S_2,s\rangle \xrightarrow{a} \langle S_2,s'\rangle$

3. If $\langle S_1,s\rangle \xrightarrow{a} \langle abort,s'\rangle$

   then $\langle S_1;S_2,s\rangle \xrightarrow{a} \langle abort,s'\rangle$

Notation

In the remainder of this thesis such rules will be written in the form

$$\frac{\text{Antecedent}}{\text{Consequent}}$$

so that, for instance, rule 1 above is written

$$\frac{\langle S_1,s\rangle \xrightarrow{a} \langle S_1',s'\rangle}{\langle S_1;S_2,s\rangle \xrightarrow{a} \langle S_1';S_2,s'\rangle}$$

□

As usual, these rules signify that if the hypothesis of the rules define transitions then the conclusions define transitions. How can these rules be interpreted? They tell us that the execution completely depends on the execution of the first statement. The configuration $\langle S_1, s\rangle$ can be transformed via transition action a in three ways; the result is either a normal configuration $\langle S_1', s'\rangle$ or a normal terminal configuration $\langle$done,s'$\rangle$ or an abnormal terminal configuration $\langle$abort,s'$\rangle$. Rule 1 says that in the first case $\langle S_1;S_2, s\rangle$ is transformed to $\langle S_1';S_2, s'\rangle$ via the same action a. Rule 2 says that in the second case $\langle S_1;S_2, s\rangle$ is transformed to $\langle S_2, s'\rangle$. Rule 3 deals with the third case and says that if the first statement aborts then so does the composition. To summarise we see that the above three rules formalise the description given in the Pascal report "The <u>compound statement</u> specifies that its component statements be executed in the same sequence as they are written" ([Jensen and Wirth 78]).

These two examples reflect the typical character of the structural approach. Two points are worth noting:

1. As with formal deductive systems of the kind employed in mathematical logic this approach defines transitions using axioms and rules. Axioms (which have no hypotheses) define the transitions directly, and rules define the transitions indirectly; that is if all hypotheses define transitions then the conclusion of the rule defines a transition. A definition of this type is called a <u>generalised inductive definition</u> . This feature makes the approach rigorously mathematical, allows a semantics to be set

up with few preconceptions, and also determines the inductive features of the proof techniques.

2. The definition of the transition relation is based on syntactic transformations of programs and simple operations on the discrete data (state and environment). So methods of proof rely heavily on structural induction which might easily be automated; and since programmers and language designers are already familiar with "symbol pushing" this form of the semantics should be more acceptable to them.

These two characteristics will become more pronounced as we progress, so for now we just summarise the above: a labelled transition system can be defined by a quadruple <C,->,A,T> and the operational semantics of a language can be given by a labelled transition system.

1.1.2. Labelled transition systems

The use of labelled transition systems to define operational semantics for concurrent programming languages is due to Plotkin [Plotkin 81] and assumes the following computational paradigm

> Processes (configurations) perform actions thereby becoming, by some transition relation, other processes. Certain processes which cannot act are regarded as having terminated .

This paradigm is embodied in the mathematical definition of a labelled transition system, a generalisation of the notion of a binary relation where a transition label (or transition action) is

associated with each pair in the relation. In fact a labelled transition system can be viewed as a labelled directed graph with a distinguished set of terminal nodes, though there have been other definitions of labelled transition systems which omit the terminal nodes.

Definition 1.1

A <u>labelled transition system</u> L is a quadruple

$$L = \langle C, \rightarrow, A, T \rangle$$

where

C	is a set of process <u>configurations</u> .
$\rightarrow$ <u>C</u> C x A x C	is a <u>labelled transition relation</u> .
A	is a set of <u>action labels</u> .
T <u>C</u> C	is a set of <u>terminal</u> configurations with the property: $t \in T$ implies $\nexists\, t', a.\ (t,a,t') \in \rightarrow$.

□

Given the strong similarity between a labelled transition system and a labelled directed graph with a distinguished set of (terminal) nodes we may describe very simple labelled transition systems by a diagram of their corresponding graphs.

A few more notational points before we proceed,

1. We may write

config(L)	for C, ranged over by c.
trans(L)	for $\rightarrow$.
labels(L)	for A, ranged over by a.
term(L)	for T.

2. $(c,a,c') \in \rightarrow$ may be written $c \xrightarrow{a} c'$, and read as

"configuration c may perform the action a, thereby becoming the configuration c' ".

It will be useful to have some definitions concerning transition relations for future use,

Definition 1.2

Let A be some arbitrary set of actions; the sets A^+ and A^* of action sequences are defined by

$$A^+ = \{(a_1,a_2,\ldots,a_n): n>0,\ a_i \in A,\ i = 1,\ldots,n\}$$

$$A^* = A^+ + \{\phi\}$$

and ranged over by u,v and w. We employ ϕ to denote the empty sequence of actions rather than the more usual ϵ since that symbol is employed later on with a different meaning.

□

Definition 1.3

Let $L = \langle C,\rightarrow,A,T\rangle$ be a labelled transition system, then

1. If $c \xrightarrow{a1} c_1 \xrightarrow{a2} c_2 \xrightarrow{a3} \ldots \xrightarrow{an} c'$

   and $w = (a_1,a_2,\ldots,a_n) \in A^+$

   or

   $w = \phi$ and $c = c'$

   then we write $c \xrightarrow{w} c'$ which may be read "the configuration c may perform the transition sequence w, thereby becoming the configuration c' ".

2. If $c \xrightarrow{w} c'$ for some $w \in A^+$ then c' is a (w-)derivative of c and we say $c \xrightarrow{w} c'$ expresses a derivation .

3. We may write

   $c \rightarrow$ for $\exists\, a,c'.\ c \xrightarrow{a} c'$

   $c \not\rightarrow$ for $\nexists\, a,c'.\ c \xrightarrow{a} c'$

$c \xrightarrow{\underline{a}}$ for $\nexists\, c'.\ c \xrightarrow{\underline{a}} c'$

$c \xrightarrow{\underline{a}\not}$ for $\nexists\, c'.\ c \xrightarrow{\underline{a}} c'$

$c \xrightarrow{\underline{W}}$ for $\nexists\, c'.\ c \xrightarrow{\underline{W}} c'$

$c \xrightarrow{\underline{a}}*$ for $\nexists\, w \in \{a\}^*.\ c \xrightarrow{\underline{W}}$

$c \xrightarrow{\underline{a}} \xrightarrow{\underline{b}} c'$ for $\nexists\, c''.\ c \xrightarrow{\underline{a}} c'' \xrightarrow{\underline{b}} c'$

This latter form will be greatly abused in its extension to multiple transition relations as $\xrightarrow{\underline{a}}_1 \xrightarrow{\underline{b}}_2$ and $\xrightarrow{\underline{a}} \xrightarrow{\underline{W}}$, and induction to $\xrightarrow{\underline{a}} \xrightarrow{\underline{b}} \xrightarrow{\underline{c}}$ etc.

4. For each $c \in C$ define the set $\underline{comp(c)}$ of computations of c by

   1. $\langle c_k \rangle_{\{1,\ldots,n\}} \in comp(c)$ if $\exists\, a_1,\ldots,a_{n-1} \in A$ such that
      $c = c_1$ and $c_1 \xrightarrow{\underline{a1}} c_2 \xrightarrow{\underline{a2}} \ldots \xrightarrow{\underline{a(n-1)}} c_n \not\rightarrow$
   2. $\langle c_k \rangle_{Nat} \in comp(c)$ if $\exists\, \langle a_j \rangle_{Nat}$ such that
      $c = c_0$ and $\forall\, k \in Nat.\ c_k \xrightarrow{\underline{ak}} a_{k+1}$.

   Thus comp(c) contains all (and only) records of maximal derivations, i.e. derivations which are either finite and cannot be extended or are infinite. We may write c_K for $\langle c_k \rangle_K$.

1.2. An example semantics for CCS

In this section we present an example semantics for the concurrent programming language CCS, and in so doing we introduce a number of definitions for use throughout this thesis. We begin by giving definitions concerning the representation and evaluation of expressions. These definitions will be considered common to all languages defined thereafter, starting with the definition of CCS which follows.

1.2.1. A common core of definitions

The languages we will define in this thesis share a common core of definitions dealing with the syntactic and semantic aspects of expressions denoting values. We present them here together with the various properties we would like them to have.

1. **Syntax**

We assume the following (disjoint unless otherwise stated) sets:

Vars a given countably infinite set $\{x_i : i \in Nat\}$ of variables, ranged over by (the metavariables) x,y,z.

Exp, a given countably infinite set of expressions ranged over by e and assumed to contain numerals for the natural numbers.

Val $\subseteq$ Exp, a non-empty set of values ranged over by v.

Bexp, a given countably infinite set of boolean expressions, ranged over by b and assumed to contain the truth values tt and ff.

It is not our intent to be concerned with the details of expressions so we merely state our assumptions and refer the reader to [Hindley et al 72] for their justification. We assume

1.1 All expressions e or b have finite sets FV(e), FV(b) $\subseteq$ Vars of free variables .

1.2 The substitution of an expression e' for a (free) variable x in an expression e or b is defined as usual, giving expressions e[e'/x], b[e'/x]. We assume that the following standard facts hold:

a) $$FV(e[e'/x]) = \begin{cases} (FV(e)\setminus\{x\})+FV(e') & \text{if } x \in FV(e) \\ FV(e) & \text{otherwise} \end{cases}$$

b) $e[e'/x][e''/x] = e[e'[e''/x]/x]$

c) $e[e'/x][e''/y] = e[e''y][e'[e''/y]/x]$ if $x \notin FV(e'')$, $x \neq y$.

2. Semantics

We will assume we are given:

States C Vars -> Val, the set of total functions, ranged over by s and $, delivering for each variable x an associated value v.

All expressions e or b can be evaluated with respect to a state s, the state providing bindings for the free variables of e or b. To this end we assume a total function as in the following definition:

Definition 1.4

We assume a total function

eval: Exp x States -> Val

and a partial function

eval: Exp -> Val

with eval(e) defined if $FV(e) = \phi$. Further, we require

eval(e,s) = eval(e[sub(X,s)])

where sub(X,s) substitutes s(x) for x for all $x \in X$. We assume similar properties for

beval: Bexp x States -> {tt,ff}, beval: Bexp -> {tt,ff}.

□

1.2.2. CCS

In a series of papers (see for example [Hennessy and Milner 85], [Milner 80],[Milner 82], and [de Nicola and Hennessy 84]), Milner

and his colleagues have studied a model of parallelism in which concurrent systems communicate by sending and receiving values along named 'channels'. Communication is synchronised in that the transmission of a value from sender to receiver takes place only when the sender and receiver are both ready, and this transmission is considered as a single event. This kind of communication, called handshake communication, is also found in a large group of modern languages such as CSP [Hoare 78] and Ada [Ichbiah et al 79].

In [Milner 80] an applicative Lambda-calculus style notation called CCS (for Calculus of Communicating Systems) is introduced for describing concurrent processes in terms of their dynamic action capability and static interconnection. More precisely there is a family of languages incorporating these ideas in various forms; in this section we present one such flavour close to that used in [Milner 80].

The language CCS, as presented here, consists of a set of operators for constructing new terms from terms which are already defined. We first give an informal description of each of these operators.

1. Inaction

'Nil' is a term representing the totally idle process which can never perform any action.

2. Action

Let AD be a set of predefined action descriptions ranged over by a, α, and assumed to include the distinguished 'silent' or

'internal' action '1'. Then 'a.' for each a ε AD is a unary (prefixed) operator, and if p is a term then 'a.p' is a term. This new term a.p represents a process which can perform the action described by a and then proceed as the process described by p.

3. Choice

If p,q are terms, 'p+q' is a term. The process represented by p+q can act either as the process represented by p or the process represented by q. The choice depends at least to a certain extent on the environment in which p+q finds itself.

4. Asynchronous parallel composition

If p,q are terms then 'p|q' is a term. This represents a composite process consisting of two sub-processes, those represented by p and q, loosely connected. The composite process performs an action when either sub-process performs an action, while its partner remains idle, or when both sub-processes simultaneously engage in complementary actions, ie one communicates a value to the other.

5. Renaming

Let Ren be a set of action renamings,ranged over by E. From each E we can define a partial function F from AD to AD such that F(1) = 1 . If p is a term then 'p[E]' is a term describing the process represented by p modified so that if p performs action a then p[E] performs F(a) if F(a) is defined. Any action a such that F(a) is not defined is inhibited.

6. Conditional execution

Assuming a set Bexp of boolean expressions ranged over by b we

introduce 'if b then p else q' as a term, whenever p and q are terms, to represent the process which acts like the process represented by p if b evaluates as being true, and like the process represented by q if b evaluates as being false.

7. Procedure call

Assuming a set Proc of procedure names ranged over by P, '$P(e_1,\ldots,e_n)$' is a term. The behaviour of the process represented by $P(e_1,\ldots,e_n)$ depends on a definition set D containing a pair of the form $(P(x_1,\ldots,x_n),p)$ where p is a term with free variables in $\{x_1,\ldots,x_n\}$. The behaviour of $P(e_1,\ldots,e_n)$ is then that of the term p with the values of $e_1,\ldots,e_n$ substituted for $x_1,\ldots,x_n$.

The syntax of CCS

The following sets are needed to define the syntax of CCS:

Act, a given countable set of action names

ranged over by A, B, and C.

Proc, a given countable set of procedure names

ranged over by P, R, and S.

Action names may also be called 'line names' or 'channels' and in later chapters we may refer to CCS with different sets of action names by subscripting e.g. CCS_{Act} .

The three main syntactic categories of CCS can now be specified as follows:

AD, the set of action descriptions, ranged

over by a, α and defined by

a ::= 1 | A!e | A?x

The intent is that

1 denotes "silent" or "internal" action.

A!e denotes the action sending the value of expression e on channel A.

A?x denotes the action receiving a value on channel A and binding it to x.

As a notational convenience we may write

A! for A!0

A? for $A?x_0$.

where x_0 is some distinguished variable occurring in terms only in these action descriptions. These forms are used when we are mainly interested in synchronisation rather than value passing. (Note x_0 is distinguished so we don't 'accidentally' bind variables).

Ren, the set of renamings, ranged over by E;

partial functions from Act to Act.

Terms, the set of terms, ranged over by p,q,r and

defined in a BNF-like notation

p ::= Nil | a.p | p+q | p|q | p[E] |

if b then p else q | $P(e_1,\ldots,e_n)$

We will allow the case n = 0 in the latter form, which will then be written P( ) or just P.

A number of notational devices will be employed, we give them here for convenient reference;

1. It will be found from the derivation rules that the following

pairs need not be distinguished:

$p+q$ and $q+p$, $(p+q)+r$ and $p+(q+r)$,

$p|q$ and $q|p$, $(p|q)|r$ and $p|(q|r)$.

We therefore allow the usual (finite) summation and product notation over indexed sets, ignoring order and association:

$$\sum_{k\in K} p_k = p_{k1} + p_{k2} + \ldots + p_{kn}$$

$$\prod_{k\in K} p_k = p_{k1} \mid p_{k2} \mid \ldots \mid p_{kn}$$

where $K = \{k_1, k_2, \ldots, k_n\}$, $n \geq 1$

2. Let $X \subseteq Act$, then

$p\backslash X$ denotes $p[E_X]$

where E_X is the renaming defined by

$$E_X(A) = \begin{cases} A & \text{if } A \notin X \\ \text{undefined otherwise} & \end{cases}$$

The process represented by $p\backslash X$ is obtained from that represented by p by inhibiting actions whose name occurs in X. As a special case of the above we define

$p\backslash A$ denotes $p\backslash\{A\}$

3. Let $A, A' \in Act$, then

$p[A'/A]$ denotes $p[E_{A,A'}]$

where $E_{A,A'}$ is the renaming defined by

$$E_{A,A'}(A'') = \begin{cases} A' & \text{if } A'' = A \\ A'' & \text{otherwise} \end{cases}$$

The process represented by $p[A'/A]$ is obtained from that represented by p by renaming the action A as A'. We will allow

$$p[A'_1, A'_2, \ldots, A'_n/A_1, A_2, \ldots, A_n]$$

to denote

$$p[A'_1/A_1][A'_2/A_2]\ldots[A'_n/A_n]$$

4. We will employ an abbreviated form of the conditional construct

if b then p to denote if b then p else Nil

A similar abbreviation for the action construct allows

a to denote a.Nil

5. To avoid excessive use of parentheses we assume the following operator precedences

[E] > a. > | > +

Thus for example

P | 1.P'\A + P"[E] means (P | (1.(P'\A))) + (P"[E])

For each term p we define the following sets

FV(p) $\subseteq$ Vars, the free variables of p.

FP(p) $\subseteq$ Proc x Nat, the free procedure names of p with their arity.

FL(p) $\subseteq$ Act, the action names mentioned in p and often referred to as sort(p) .

in a tabular fashion (We will assume that for (P,n) ε Defined(D) we have a set FL(P,n) $\subseteq$ Act which defines its sort).

	Nil	1.p	A!e.p	A?x.p	p+q
FV	ϕ	FV(p)	FV(e)+FV(p)	FV(p)-{x}	FV(p)+FV(q)
FP	ϕ	FP(p)	FP(p)	FP(p)	FP(p)+FP(q)
FL	ϕ	FL(p)	FL(p)+{A}	FL(p)+{A}	FL(p)+FL(q)

	p|q	p[E]	$P(e_1,\ldots,e_n)$
FV	FV(p)+FV(q)	FV(p)	$FV(e_1)+\ldots+FV(e_n)$
FP	FP(p)+FP(q)	FP(p)	{(P,n)}
FL	FL(p)+FL(q)	E(FL(p))	FL(P,n)

where E(FL(p)) denotes the image under E of FL(p).

	if b then p else q
FV	FV(p)+FV(q)+FV(b)
FP	FP(p)+FP(q)
FL	FL(p)+FL(q)

Note we also assume the substitution of expressions for variables in expressions is extended in the usual way to allow the substitution of expressions for variables in terms, written p[e/x], and the substitution of closed terms for parameterless procedure calls P(), written p[q/P]. (Of course in the former case such substitutions may require a change of bound variables to avoid clashes).

Finally, a <u>definition set</u> D is a partial function

$$D:\{P(x_1,\ldots,x_n):\ P \in Proc,\ x_1\ldots x_n \in Vars\} \rightarrow Terms$$

such that if $D(P(x_1,\ldots,x_n)) = p$ is defined then,

1. $FV(p) \subseteq \{x_1,\ldots,x_n\}$
2. $FP(p) \subseteq Defined(D)$, where

   $Defined(D) = \{(p,n):\ \exists\, x_1\ldots x_n.\ \ D(P(x_1,\ldots,x_n))$ is defined$\}$
3. $x_i \neq x_j$ if $i \neq j$.
4. $FL(p) \subseteq FL(P,n)$.

We will assume that

$$D(P(x_1,\ldots,x_n)) = p,\ D(P(y_1,\ldots,y_n)) = q$$

implies

$$p = q,\ y_i = x_i \quad i = 1,\ldots,n$$

so that (P,n) uniquely identifies the definition.

To describe a definition set D we introduce the following notation for definitions:

$$P(x_1,\ldots,x_n) \Leftarrow p \text{ denotes } ((P(x_1,\ldots,x_n),p) \in D$$

We shall impose a slight constraint on the collection D, forbidding definition sets containing such definitions as

$$P(x) \Leftarrow A!x + P(x+1)$$

or

$$\begin{cases} P(x) \Leftarrow P'(x) \\ P'(x) \Leftarrow P(x) \end{cases}$$

in which a process may "call itself recursively without first performing an action". Thus the following are permitted

$$P(x) \Leftarrow A!x + 1.P(x+1)$$

or

$$\begin{cases} P(x) \Leftarrow P'(x) \\ P'(x) \Leftarrow 1.P(x) \end{cases}$$

We won't here go into the details of how such guardedly well-defined definition sets are formally defined, instead we refer the reader to [Milner 80].

The operational semantics of CCS

To define the operational semantics of CCS we define a labelled transition system which must of course be parameterised on the syntactic parameters Act and Proc. Further, in giving a semantics

for terms of the form $P(e_1,\ldots,e_n)$ we need to refer to a definition set D. Thus our labelled transition system should properly be written with three subscripts. We eschew this notational horror, dropping all subscripts unless one or more is of interest, whenever no confusion can arise.

Now assume we have sets Act, Proc and D as described previously, we define the labelled transition system L[CCS] by

$$\text{config}(L[CCS]) = \{p:\ FV(p)=\phi,\ FP(p) \subseteq \text{Defined}(D)\}$$

$$\text{labels}(L[CCS]) = \{\varepsilon, A!v, A?v:\ A \in Act,\ v \in Val\}$$

$$\text{term}(L[CCS]) = \phi$$

Thus configurations are terms with no free variables and no undefined procedure calls. It is common when defining the configurations of a labelled transition system to define a predicate '$\vdash$' signifying <u>syntactic validity</u> over some set of potential programs. Thus in this case

$$\vdash p \text{ iff } FV(p) = \phi,\ FP(p) \subseteq \text{Defined}(D),\ p \in \text{Terms}.$$

The definition of this predicate is often referred to as a <u>static semantics</u> since it deals with syntactic properties of the program text. In contrast we now turn, in the definition of trans(L[CCS]) to the <u>dynamic semantics</u> of (valid) programs.

To aid in the description of handshake communication (rule 4.3 to follow) we formalise a notion of <u>complementary action</u>. Two action labels a and $\bar{a}$ are complementary if one describes the sending of a value on some channel and the other describes the receipt of the same value on the same channel. The simultaneous performance of complementary actions by parallel processes

constitutes a handshake communication. So define

$$\bar{a} = \begin{cases} A!v & \text{if } a = A?v \\ A?v & \text{if } a = A!v \\ \text{undefined} & \text{otherwise} \end{cases}$$

In order to define trans(L[CCS]) we first define a more primitive transition relation -> which deals explicitly with the internal actions, denoted by 1, of a process description. Then trans(L[CCS]) will be defined so as to ignore internal 1 actions. The transition relation -> (not trans(L[CCS])) is defined by

1. Inaction

   Nil has no transitions.

2. Action

   2.1 $1.p \xrightarrow{1} p$

   2.2 $$\frac{eval(e)=v}{A!e.p \xrightarrow{A!v} p}$$

   2.3 $A?x.p \xrightarrow{A?v} p[v/x]$

3. Choice

   3.1 $$\frac{p \xrightarrow{a} r}{p+q \xrightarrow{a} r}$$ 3.2 $$\frac{q \xrightarrow{a} r}{p+q \xrightarrow{a} r}$$

4. Parallel

   4.1 $$\frac{p \xrightarrow{a} r}{p|q \xrightarrow{a} r|q}$$ 4.2 $$\frac{q \xrightarrow{a} r}{p|q \xrightarrow{a} p|r}$$

   4.3 $$\frac{p \xrightarrow{a} p' \ , \ q \xrightarrow{\bar{a}} q'}{p|q \xrightarrow{1} p'|q'}$$

5. Renaming

5.1 $$\frac{p \xrightarrow{a} q}{p[E] \xrightarrow{F[a]} q[E]}$$

if F(a) is defined where

$$F(a) = \begin{cases} 1 & \text{if } a = 1 \\ E(A)!v & \text{if } a = A!v,\ E(A) \text{ defined} \\ E(A)?v & \text{if } a = A?v,\ E(A) \text{ defined} \\ \text{undefined otherwise} & \end{cases}$$

6. Conditional

6.1 $$\frac{beval(b) = tt\ ,\ p \xrightarrow{a} r}{\text{if } b \text{ then } p \text{ else } q \xrightarrow{a} r}$$

6.2 $$\frac{beval(b) = ff\ ,\ q \xrightarrow{a} r}{\text{if } b \text{ then } p \text{ else } q \xrightarrow{a} r}$$

7. Procedure call

7.1 Suppose $D(P(x_1,\ldots,x_n)) = p$ is defined then

$$\frac{p[v_1,\ldots,v_n/x_1,\ldots,x_n] \xrightarrow{a} r}{P(e_1,\ldots,e_n) \xrightarrow{a} r}$$

if $eval(e_i) = v_i$ for $i = 1,\ldots,n$.

Now, having defined ->, we can define

$trans(L[CCS]) = \Rightarrow$

by ignoring the internal 1 actions of -> as follows:

$p \xRightarrow{\varepsilon} q$ if $p \xrightarrow{w} q$, $w \in \{1\}^*$

$p \xRightarrow{a} q$ if $a \in labels(L[CCS])$, $p \xRightarrow{\varepsilon} \xrightarrow{a} \xRightarrow{\varepsilon} q$

In keeping with the intended interpretation of 1 and ε as 'silent'

action we may often drop their occurence as action labels, writing

-> for $\xrightarrow{1}$, and => for $\xRightarrow{\varepsilon}$

An example

To demonstrate most of the features of CCS we consider a very simple CCS program to compute the n'th fibonacci number, fib(n), defined $\forall$ n $\geq$ 0 by

fib(0) = 1

fib(1) = 1

fib(x) = fib(x-1) + fib(x-2) if x $\geq$ 2

We would like to define a CCS agent 'fibber(x)' that behaves as if defined by:

fibber(x) <= result!fib(x)

Consider the agent defined by:

```
fibber(x) <=
        if   x < 2
        then result!1
        else ((fibber(x-1)|fibber(x-2))[subresult/result] |
              subresult?x.subresult?y.result!x+y
             )\subresult
```

The idea is that if x < 2 then fibber(x) knows the result is 1 and can signal that fact and stop. Otherwise it starts up two subprocesses, fibber(x-1) and fibber(x-2), in parallel to calculate fib(x-1) and fib(x-2). To prevent these sub-agents signalling their results to anything other than their caller their outputs are renamed to be subresults and restricted so they can only be accessed by their caller. The caller receives these two subresults and binds them in the variables x and y, finally signalling their sum as the result.

To see how the operational semantics works consider a possible first step of fibber(2):

$$\text{fibber}(2) \xrightarrow{1} ((\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}] \mid \text{subresult?y.result!x+y})\backslash\text{subresult}$$

because by rule 2.2

$$\text{result!1.Nil} \xrightarrow{\text{result!1}} \text{Nil}$$

so by rule 6.1

if 0 < 2

then result!1.Nil

else ((fibber(0-1)|fibber(0-2))[subresult/result] | subresult?x.subresult?y.result!x+y)\subresult

$\xrightarrow{\text{result!1}}$

Nil

so by rule 7.1

$$\text{fibber}(2-2) \xrightarrow{\text{result!1}} \text{Nil}$$

so by rule 4.2

$$\text{fibber}(2-1)|\text{fibber}(2-2) \xrightarrow{\text{result!1}} \text{fibber}(2-1)|\text{Nil}$$

so by rule 5.1

$$(\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult/result}] \xrightarrow{\text{subresult!1}}$$

$$(\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}]$$

so by rule 4.3

$$((\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult/result}] \mid \text{subresult?x.subresult?y.result!x+y}) \xrightarrow{1}$$

$$((\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}] \mid$$

```
     subresult?y.result!1+y)
so by rule 5.1
   ((fibber(2-1)|fibber(2-2))[subresult/result] |
    subresult?x.subresult?y.result!x+y
   )\subresult
 -1->
    ((fibber(2-1)|Nil)[subresult/result] |
     subresult?y.result!1+y
    )\subresult
so by rule 6.2
    if   2 < 2
    then result!1
    else ((fibber(2-1)|fibber(2-2))[subresult/result] |
          subresult?x.subresult?y.result!x+y
         )\subresult
  -1->
    ((fibber(2-1)|Nil)[subresult/result] |
     subresult?y.result!1+y
    )\subresult
whence, finally, by rule 7.1
   fibber(2) -1-> ((fibber(2-1)|Nil)[subresult/result] |
                   subresult?y.result!1+y
                  )\subresult
The remaining steps of the computation are
   ((fibber(2-1)|Nil)[subresult/result] |
    subresult?y.result!1+y
   )\subresult
 -1->
```

```
((Nil|Nil)[subresult/result] |
 result!1+1
)\subresult
_result!2>
((Nil|Nil)[subresult/result] |
 Nil
)\subresult
= p (say)
```

If now we use the transition relation => to ignore $\xrightarrow{1}$ moves resulting from the internal communications of subresults we get

$$fibber(2) \xrightarrow{result!2} p$$

and p can perform no further actions.

2. A bisimulation theory of translation correctness

The aim of this chapter is to describe a theory of translation correctness for programming languages whose semantics is given in the abstract operational manner i.e. by a labelled transition system. Any proposed theory of translation correctness will be judged on many criteria, so perhaps it is best to start with a brief examination of the more obvious ones.

Firstly, of course, the theory must be general enough to apply to a large number of interesting translations while being sufficiently powerful to say something interesting about the properties of those translations. In general this must be a trade-off, the more powerful the statement of correctness given by the theory the fewer the number of translations likely to satisfy those conditions. Conversely a surfeit of generality can easily lead to a rather anaemic set of correctness criteria.

Secondly, for the purposes of clarity and modularity in the translations to which the theory is applied it would be greatly advantageous to have a theory which is <u>compositional</u> in the sense that the correctness of a translation which incorporates other sub-translations can be proved assuming only that the sub-translations have been proved correct in the sense of the theory. Given that the theory supports such a proof method the writing of translations themselves can be made easier by segmenting the problem into pieces, a large translation can be written in terms of other translations for subcomponents, and the specified correctness of the whole can be proved given any subtranslations which satisfy the correctness criteria.

Thirdly, a simple intuitive statement of correctness is to be desired so that it is easily seen what properties the translation has; criteria couched in terms of obscure mathematical concepts about the theory of computation will not be readily acceptable to most computer scientists.

Lastly, given the seemingly inherent complexity of most translations and their correctness proofs it would be useful if there were some hope of automating at least a proof checker, and preferably a proof generator.

The basic approach of this chapter will be to generalise a notion of behavioural equivalence which has been previously used in proving the equivalence of process descriptions in CCS [Milner 82]. In the previous work the process descriptions were configurations in the same transition system. Here we will depart from this approach in that we will need to extend the notion of equivalence to objects in different transition systems. This will require some refinement of the definitions but the major difference will lie in the addition of a non-behavioural component to the equivalence in order to cope simply with the notion of final states. We believe that the theory to be outlined goes some way to achieving each of the above criteria.

2.1.1. Review

As we saw in the introduction there have been a number of previous attempts to outline a theory of translation for programming languages, the most readily accepted of which is the denotational theory of Morris [Morris 73] which is described in terms of

commuting diagrams between algebras. This is a very elegant and intuitive theory but for our purposes it fails at an important point. Our attempt is aimed mainly for programming languages with non-determinism and parallelism built in as primitives; as yet there is no satisfactory treatment of such features within the algebraic approach.

Instead we turn to work already performed in the area of operational semantics by essentially two groups; in order to focus the discussion we will examine a single piece of work from each with particular reference to the criteria above rather than the less technical criteria discussed in the introduction.

The first, and most comprehensive, piece is the thesis of Wei Li [Li 83]. Chapter 5 of his thesis details an operational translation theory with much the same intent as this work. Roughly speaking the basic ideas of his approach were:

1. A <u>syntactic translation</u> between languages $Lang_1$ and $Lang_2$ is a function f: $Lang_1$ -> $Lang_2$ translating programs in $Lang_1$ to programs in $Lang_2$. Any syntactic translation induces a semantic translation between the transition systems which define ,in the structural axiomatic form, the operational semantics of $Lang_1$ and $Lang_2$. Configurations in the respective transition systems will typically be composed of programs and some semantic elements such as states or environments. A <u>semantic translation</u> is a function from configurations to configurations which translates each component separately i.e. programs to programs and semantic components to semantic components. This is the view we will

take, and in the remainder of this thesis we will assume this distinction.

2. The execution of a program together with an input state can be represented by a finite transition sequence ending in a terminal configuration (successful computation), by a finite transition sequence ending in a nonterminal configuration (deadlocked computation) , or by an infinite transition sequence (divergent computation). Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the object system and its translation in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program for any given initial input state.

3. Having formalised the idea of the correctness of a translation, sufficient conditions, called adequacy conditions, which guarantee the correctness of a translation are proposed. It is adequacy which is established in the body of Li's proofs, correctness follows as a corollary.

This theory possesses substantial generality and power but falls down in some of its details. For instance,

a. The requirement that termination and deadlock be distinguished within and between transition systems is just too stringent for some translations ; in his example translation from CSP to CCS Li needed to doctor the CCS

transition system definition to introduce an appropriate distinction between deadlock and termination. By the nature (cleanliness) of his translation he was able to determine the appropriate choice of terminals in a way which may not be possible in general.

b. The compositionality of his theory is very weak as it is double layered; a correctness criterion together with an underlying set of conditions sufficient to guarantee correctness. A composition theorem can be proved for correctness [Li 83 Theorem 5.2] but when trying to prove a translation with sub-translations correct the correctness of the sub-translations is unlikely to enable proof of the adequacy conditions to support correctness of the complete translation. A solution would be to prove a composition theorem for adequacy but as yet this has not been achieved.

It should be noted that adequacy is a good deal stronger than correctness, dealing as it does with the observable actions of the processes as well as their resulting states, and thus is not a set of conditions necessary for correctness.

c. The statement of correctness is quite clear but says little about behaviour while the statement of adequacy says a good deal about behaviour but is rather opaque.

d. The proof technique of establishing adequacy leads to a very large detailed proof holding little hope for machine checkability. While the former charge may be laid against almost any translation proof, including those to follow unfortunately, the level of detail and subtlety is sufficient

to make the proofs very difficult to follow.

The second theory advanced is contained in a paper by Astesiano and Zucca [Astesiano and Zucca 82], wherein again the translation example is CSP to CCS. The approach taken is quite similar to that proposed in this chapter in that an existing equivalence for CCS is extended to a theory of translation. To introduce such non-behavioural factors as final state equality they resort to the introduction of processes that write out the contents of the final state so that the states can be compared in a behavioural way. This introduces processes into the translation which are there just for the theory to work rather than for the translation to work. The theory proposed here takes into account factors other than just behavioural properties of the processes and thus avoids the introduction of unnecessary terms.

The equivalence generalised in that work was the observational equivalence proposed by Milner [Milner 80]; the equivalence generalised in this chapter is the bisimulation equivalence of Park [Park 81b],[Milner 82]. This more recent equivalence has the advantage of an elegant proof technique while delivering a slightly finer equivalence than observational equivalence. This proof technique holds out some hope of machine implementation, programs to check simple bisimulations may easily be written in a number of programming languages. However a great deal more work must be done before bisimulations in the form used in the proof of chapter 3 can be generated by machines.

2.1.2. Bisimulations

In this subsection we review the notion of bisimulation equivalence between processes and the proof technique that the theory affords. The essence of the approach is that we do not wish to distinguish agents which have the same derivation tree, this is made precise by the notion of bisimulation between agents.

Definition 2.1

Let $L = \langle C, \rightarrow, A, T \rangle$, then for any $R \subseteq C \times C$ define $F(R)$ to be the set of pairs (c_1, c_2) satisfying

1. $c_1 \xrightarrow{a} c_1'$ implies $\exists\, c_2' \,.\, c_2 \xrightarrow{a} c_2' \;\&\; c_1' \; R \; c_2'$
2. $c_2 \xrightarrow{a} c_2'$ implies $\exists\, c_1' \,.\, c_1 \xrightarrow{a} c_1' \;\&\; c_1' \; R \; c_2'$

whenever $a \in A$. A <u>bisimulation</u> is a relation $R \subseteq C \times C$ such that $R \subseteq F(R)$, thus R is a bisimulation iff 1 and 2 above hold for R. Two configurations $c_1, c_2 \in C$ are (bisimulation) <u>equivalent</u>, written $c_1 \sim c_2$, if there exists a bisimulation R with $c_1 \; R \; c_2$.

□

It should be remembered that F depends on the particular transition relation involved and so should most properly be written $F_{\rightarrow}$. However, we will generally omit the subscripts for convenience.

The motivation behind this definition may be made clearer by a simple paraphrase of the definition (given already in the introduction):

> "Two agents are equivalent if whenever one of them can perform an observable action so can the other in such a way

that the resulting agents are equivalent".

Now we have an elegant proof technique; to show $c_1 \sim c_2$ it is necessary and sufficient to find a bisimulation containing the pair (c_1, c_2). To demonstrate the proof technique we will consider an example from CCS.

Proposition 2.1

$(1.wave!)|shout! \sim 1.(shout!|wave!)$

Proof: It is sufficient to provide a bisimulation containing the above pair of agents; the following set R of pairs is just such a relation.

$$\{ (\ 1.wave!|shout!\ ,\ 1.(shout!|wave!)\ ), \quad [1]$$
$$(\ wave!|shout!\ ,\ shout!|wave!\ ), \quad [2]$$
$$(\ Nil|shout!\ ,\ shout!|Nil\ ), \quad [3]$$
$$(\ 1.wave!|Nil\ ,\ Nil|wave!\ ), \quad [4]$$
$$(\ wave!|Nil\ ,\ Nil|wave!\ ), \quad [5]$$
$$(\ Nil|Nil\ ,\ Nil|Nil\ ) \quad [6]$$
$$\}$$

To prove R is a bisimulation we will apply the following Lemma which helps reduce the number of cases which must be considered.

Lemma 2.1

Suppose Rel $\subseteq$ Terms x Terms, then Rel $\subseteq F_{\Rightarrow}(\text{Rel})$ is implied by the following closure condition: Whenever p Rel q all the following hold,

1.1 If $p \xrightarrow{1} p'$ then $\exists\, q'.\ q \stackrel{\varepsilon}{\Rightarrow} q'$, p' Rel q'

1.2 If $p \xrightarrow{a} p'$, $a \neq 1$ then $\exists\, q'.\ q \stackrel{a}{\Rightarrow} q'$, p' Rel q'

2.1 If $q \xrightarrow{1} q'$ then $\exists\, p'.\ p \stackrel{\varepsilon}{\Longrightarrow} p'$, p' Rel q'

2.2 If $q \xrightarrow{a} q'$, $a \neq 1$ then $\exists\, p'.\ p \stackrel{a}{\Longrightarrow} p'$, p' Rel q'

Proof by induction on the length of the derivation $p \stackrel{a}{\Longrightarrow} p'$ or $q \stackrel{a}{\Longrightarrow} q'$.

□

To apply this lemma we take each pair (p,q) in Rel and establish conditions 1.1 ... 2.2, first by considering 1.1 and 1.2 , and then considering what amounts to the converse, i.e. 2.1 and 2.2. Consider establishing condition 1.1 for pairs in [1], this will be written

$$[1] \xrightarrow{i} [2]$$

$$[1] \xrightarrow{\text{shout!}} [4]$$

This is intended to express the statement

If $(p,q) \in [1]$ and $p \xrightarrow{a} p'$ then

either $a = 1$, $\exists\, q'.\ q \stackrel{\varepsilon}{\Longrightarrow} q'$, $(p',q') \in [2]$

or $a = \text{shout!}$, $\exists\, q'.\ q \stackrel{\text{shout!}}{\Longrightarrow} q'$, $(p',q') \in [4]$

Using this notation the proof may proceed by cases on [n] for each $n \in \{1,\ldots,6\}$, each case being considered first for condition 1 and then condition 2:

[1] First 1: $[1] \xrightarrow{1} [2]$

$[1] \xrightarrow{\text{shout!}} [4]$

Then 2: $[1] \xrightarrow{1} [2]$

[2] First 1: $[2] \xrightarrow{\text{wave!}} [3]$

$[2] \xrightarrow{\text{shout!}} [5]$

Then 2: $[2] \xrightarrow{\text{wave!}} [3]$

[2] -shout!-> [5]

[3] First 1: [3] -shout!-> [6]

Then 2: [3] -shout!-> [6]

[4] First 1: [4] -1-> [5]

Then 2: [4] -wave!-> [6]

[5] First 1: [5] -wave!-> [6]

Then 2: [5] -wave!-> [6]

□

2.1.3. Problems with bisimulations

The definition of bisimulation equivalence is based on a completely observational notion of comparison by visible action; properties of the agents which do not show up as observational differences will not be distinguished. This is not to say that the theory is not extendable by the introduction of more powerful observers who can distinguish hitherto equivalent agents, rather that having selected a form of observation it may also be found useful to examine agents without completely specifying how the examination is to be performed. We will return to this approach in defining P-bisimulations but for now we will consider a few examples of agents which are bisimulationally equivalent but differ in certain interesting properties.

Example 1: The equivalence of non-movers.

In a concurrent setting such as CCS, configurations may have no derivatives (i.e. be unable to proceed) for one of two reasons. Firstly they may be configurations representing a successful conclusion to some computation, an example of this might be Nil in

CCS. The second reason involves the concept of deadlocked configurations wherein, in a simple case, two agents are attempting to proceed with some activity yet are unable to synchronise their efforts. For example if two overly polite people reach a door at the same time each may wait for the other to pass first, with the result that neither can pass. This is called a deadlock and can be modelled in CCS by the configuration

fred_first!.bill_first? || bill_first!.fred_first?

In this analogy Nil || Nil might model a state where neither agent wants to pass through the door, both are content where they are. However, we have

fred_first!.bill_first? || bill_first!.fred_first? ~ Nil||Nil

which fails to recognise that one term (the left side) is 'frustrated' and the other 'content' in our interpretation. More formally may we say one is deadlocked and the other terminal.

Example 2: Silent spinning

The statement of bisimulation equivalence is given in terms of the relation -> which is ternary; $c_1 \xrightarrow{a} c_2$ is to be interpreted as asserting that the agent c_2 is a result of agent c_1 performing action a. This fails to capture the notion of an infinite sequence of 'silent' actions (say 1 moves in CCS with => as the transition relation) wherein there is no result agent; infinite silent action is simply not accounted for by the definitions. For example if we assume the definition

spin <= 1.spin

then, with respect to the => transition relation, Nil and spin have (up to renaming) identical derivations, ie

Nil $\xRightarrow{\varepsilon}$ Nil only, [1]

and spin $\stackrel{\varepsilon}{\Rightarrow}$ spin only. [2]

Thus => fails to distinguish between transitions wherein no action is performed,such as [1], and transitions, such as [2], wherein a silent action may be performed. This leads to

spin = Nil

and spin|a! = a! [3]

which is intuitively unpleasant since in [3] the agent on the left need never perform the action a!.

2.1.4. Translation correctness

We can now define a (semantic) translation as a function from configurations to configurations; such a function may well be induced by a syntactic translation in the manner discussed earlier but this is not necessary to the definition. In fact the definition could allow partial functions as translations but for simplicity we adopt the stronger form. In what follows let L_1 = $\langle C_1, A, \rightarrow_1, T_1 \rangle$, L_2 = $\langle C_2, A, \rightarrow_2, T_2 \rangle$ be two transition systems over the same set of actions. Where no confusion can occur we will simply drop the subscripts from the transition relations.

Definition 2.2

A <u>translation</u> from L_1 to L_2 is a total function tr: C_1 -> C_2 and we may refer to L_1 as the <u>source</u> system and L_2 as the <u>target</u> system .

□

To handle translations the theory of bisimulations will be extended to the case where the agents related are configurations in different transition systems. For simplicity we will assume

that the two transition systems have the same set of actions, though it would be quite simple to relax this requirement it would also be notationally disastrous.

Definition 2.3

We first extend F as given previously in Definition 2.1; for any R $\underline{C}$ $C_1 \times C_2$ define F(R) to be the set of pairs (c_1,c_2) satisfying

1. $c_1 \xrightarrow{a}_1 c_1'$ implies $\exists\, c_2' \,.\, c_2 \xrightarrow{a}_2 c_2'$, c_1' R c_2'
2. $c_2 \xrightarrow{a}_2 c_2'$ implies $\exists\, c_1' \,.\, c_1 \xrightarrow{a}_1 c_1'$, c_1' R c_2'

whenever $a \in A$.

A bisimulation is a relation R $\underline{C}$ $C_1 \times C_2$ such that R $\underline{C}$ F(R). Thus the case when $L_1 = L_2$ is just the usual definition.

□

Turning to the criteria for translation correctness we introduce the notion of <u>good</u> translations as translations which preserve the behaviour of the source configuration as the behaviour of its image, simply by demanding there exists a bisimulation containing the pair (c,tr(c)) for each source configuration c.

Definition 2.4

A translation tr from L_1 to L_2 is <u>good</u> if there exists a bisimulation R $\underline{C}$ $C_1 \times C_2$ such that $\forall\, c \in C_1 .$ c R tr(c)

□

Having generalised bisimulations to pairs of transition systems we now add an extra discriminatory mechanism, a P-bisimulation R is a bisimulation such that c_1 R c_2 implies c_1 P c_2.

Definition 2.5

Let P $\underline{C}$ $C_1 \times C_2$ be an arbitrary binary predicate. A <u>P-bisimulation</u> is a bisimulation R $\underline{C}$ P.

□

Of course we can now augment Definition 2.4:

Definition 2.6

A translation tr is <u>P-good</u> if there exists a P-bisimulation R such that

$$\forall\ c \in C_1\ .\ c\ R\ tr(c)$$

□

To prove a translation tr is P-good we first demonstrate a bisimulation between configurations of the object system and their translations, then we show that the bisimulation is wholly contained in P. Thus with this definition of translation correctness we can retain the normal bisimulation proof technique in a slightly augmented form.

One of the properties we would most like to have is compositionality,

Proposition 2.3

Suppose

$L_i = \langle C_i, \rightarrow_i, A, T_i \rangle \quad i = 1,2,3,$

$P_i \subseteq C_i \times C_{i+1} \quad i = 1,2,$

$P_3 \subseteq C_1 \times C_3,$

$tr_i: L_i \rightarrow L_{i+1}$ is P_i-good $i = 1,2,$

$c_1\ P_1\ c_2,\ c_2\ P_2\ c_3$ implies $c_1\ P_3\ c_3$

then

$tr_2 \circ tr_1$ is P_3-good

Proof we have two bisimulations

$R_i \subseteq P_i,\ \forall\ c_i \in C_i .\ c_i\ R_i\ tr_i(c_i)\ ,\ i=1,2$

the take

$$R_3 = \{(c_1,c_3): \exists\, c_2.\ (c_1,c_2) \in R_1,\ (c_2,c_3) \in R_2\}$$
$$\underline{C}\ P_3$$

□

To justify the introduction of the predicate P we now show its utility in a few simple examples

Examples of non-behavioural predicates

We have already seen in the review of bisimulations that we would like a few 'standard' ways of extending the notion of bisimulation; by varying P in the definition of P-goodness we can alter the requirements for translation correctness to capture these extensions. Note it is not being suggested that the following examples for P should always be used, for instance the main example in the next chapter is not terminal-good, where 'terminal' is as defined in example 1 to follow. The intent is to allow a degree of flexibility in the translation correctness requirements while retaining a general framework within which many examples will fit.

Example 1

c_1 **terminal** c_2 iff $(c_1 \in T_1$ iff $c_2 \in T_2)$

Thus a terminal-bisimulation cannot confuse deadlocked and terminal states; terminals may only be paired with terminals.

Example 2

There are many ways in which divergence might be defined to indicate properties of agents such as the ability to 'spin' silently or some degree of syntactic 'under-definedness'. Exactly how it is defined will depend on the transition system and the

potentially undesirable properties of certain configurations in that system, however suppose we have predicates $div_1 \subseteq C_1$ and $div_2 \subseteq C_2$ which indicate divergence of their arguments. Then we can define

c_1 **converge** c_2 iff $(div_1(c_1)$ iff $div_2(c_2))$

A converge-bisimulation existing between two agents implies that each can diverge after a sequence of actions only if the other can.

Example 3

Suppose we have two total valuation functions $f_1: C_1 \rightarrow V$ and $f_2: C_2 \rightarrow V$ for some set V of values, then define

c_1 **valuation** c_2 iff $f_1(c_1) = f_2(c_2)$

This sort of relation allows the comparison of non-behavioural properties of configurations; for example a comparison of states in CSP or imperative languages in general.

2.1.5. Remarks

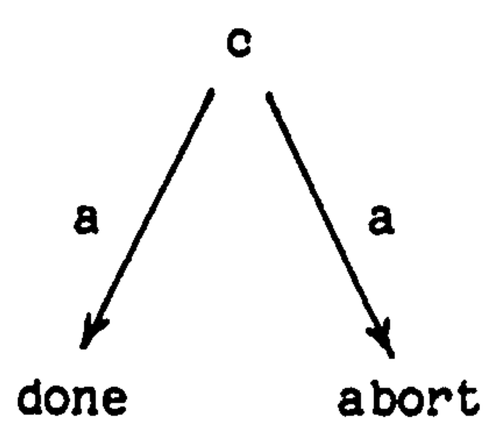
We will now examine a few simple consequences of the definition of P-goodness and point out one of the peculiarities of this approach.

Proposition 2.2

If R is a P-bisimulation and c_1 R c_2 it does not follow that every bisimulation containing the pair (c_1, c_2) is a P-bisimulation.

Proof: Recalling the graphical notation for transition systems introduced in Chapter 1 , take

$L_1 = L_2 =$



$P = \{\langle c,c\rangle , \langle done,done\rangle , \langle abort,abort\rangle\}$

Then P is a P-bisimulation with what is probably the intended matching of terminals. However the following is also a bisimulation but not a P-bisimulation:

$Q = \{\langle c,c\rangle , \langle done , abort\rangle , \langle abort , done\rangle\}$

□

At first sight this looks rather serious,implying essentially that in some cases the P in P-goodness seems to have no restrictive power over the correspondence between source and target configurations. The intent in introducing P was to establish 'allowable' correspondence between source and target configurations, the idea being that after any sequence of actions performed by both a source configuration and its translation the resulting configurations should be related by P. For instance, if P was a valuation comparison of states then we would expect that the final configurations have the same state component. The example above shows this is not necessarily true and we now examine what, if anything, has been lost.

The following proposition asserts that for every finite computation of source (or target) there is a corresponding finite computation of target (or source). Further, if the source (target) computation has a single outcome then all outcomes of the target

(source) are P-related to that outcome.

Proposition 2.4

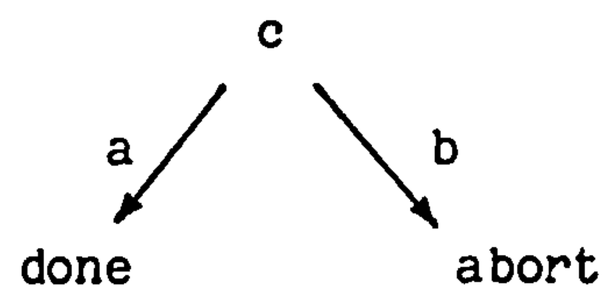
Suppose R is a P-bisimulation, c_1 R c_2 , then $\forall$ w $\in$ A^*

1. If $c_1 \stackrel{w}{\Longrightarrow} c_1'$ then $\exists\, c_2' \,.\, c_2 \stackrel{w}{\Longrightarrow} c_2'$, c_1' R c_2'
2. If $c_2 \stackrel{w}{\Longrightarrow} c_2'$ then $\exists\, c_1' \,.\, c_1 \stackrel{w}{\Longrightarrow} c_1'$, c_1' R c_2'
3. If $c_1 \stackrel{w}{\Longrightarrow} c_1'$ and $(c_1 \stackrel{w}{\Longrightarrow} c$ implies $c = c_1')$ then
   $\forall\, c_2' \,.\, c_2 \stackrel{w}{\Longrightarrow} c_2'$ implies c_1' P c_2'

Proof by induction on the length of the derivation of $\stackrel{w}{\Longrightarrow}$.

□

Thus if both source and target run through a sequence w of actions then the resulting configurations are not <u>necessarily</u> related in P unless w "specifies to within P" those configurations. For instance in the example above a can lead to either done or abort, it does not specify which. However, if we changed the system to



where a ≠ b then P is the only bisimulation.

Having now outlined the theory it seems appropriate to consider some of its features and faults. The first point to note is that no restriction has been placed on the form that P may take. Thus in transition systems with a rather impoverished set of actions little of the work of comparison may be done by the bisimulation part of the definition and much more by the P part. In the extreme case where no externally observable actions are performed all the descriminatory power lies with P. Since this theory was

not really aimed at such systems this is not a cause of great concern, but it does lead to certain theoretical difficulties as to the exact power of the theory. The theory is at least as strong as a pure bisimulation theory but beyond that it seems to have few limits. This gives it more the flavour of a theory schema, which at least can be defended on the grounds of flexibility, a framework within which many different sorts of translations can be proved correct. In the next chapter we articulate the theory on an example translation in which subtranslations carry in their P predicates properties that are necessary for their correct functioning within their intended environment. These extra properties are highly dependent on the nature of the particular translation which requires them and thus have no place in the statement of a general theory.

Secondly, the efficacy of the theory relies a great deal on an appropriate choice of transition systems. The intended interpretation of the bisimulation relation is that of some observer performing tests on the external observable actions of two systems and comparing the results. If the ternary -> relation does not match the intuitions of observability then some degree of intuitive acceptability may be lost.

Lastly, as we have noted, this theory is at least as strong as a purely bisimulational theory. It may be that for some commonly occurring translations this is rather too strong already, there may be intuitively acceptable translations which are not P-good for any P. This is a point which we will return to in later chapters but for the moment we will just remark on it in passing.

3. An example translation in the bisimulation theory

The aim of this chapter is to present an improvement of the translation given in [Li 83 chapter 6]. The improvement takes two forms; firstly, the translation is extended to cover a larger range of syntactic forms. Specifically, the restriction of [Li 83 chapter 6 section 6.2.2] on the allowable forms of guarded commands is dropped so that the form of guarding is the more general 'strong' form for all commands. Secondly, an improved proof technique is obtained by using the bisimulation theory of the previous chapter.

The description of CSP employed is essentially that of [Li 83 chapter 2] with some minor changes to simplify the statement of correctness; these changes will be pointed out and explained as they occur. Due to the great bulk of the description of CSP and the large degree of case analysis inherent in the bisimulation approach, only the most difficult and exemplary cases are dealt with in the proof in order that this chapter maintain a reasonable size. Section 3.1 constitutes a formal syntax and semantics for the variant of CSP we will employ, section 3.2 introduces the translation, section 3.3 provides the correctness criteria, and section 3.4 establishes the claimed correctness.

3.1. A variant of CSP

In common with all languages defined in this thesis our variant of CSP shares the definitions, given in chapter 1, of expressions, values ,states etc. and in what follows we take these as given. We will closely follow the description given in [Li 83 chapter 2]

since our interest is not, primarily, in providing a new semantics for CSP; we take that as having been achieved and refer the interested reader to [Li 83 chapter 2] whenever additional motivation or explanation is sought. As usual we begin with the syntax;

3.1.1. Syntax

The syntax of CSP is parameterised on the defined categories of section 1.2.1 and the following syntactic categories:

Plab - a given countably infinite set of **process labels**,
ranged over by P, Q, R.

Pten - a given countably infinite set of **pattern symbols**,
ranged over by W.

The categories of guarded commands and commands of CSP are defined using a BNF-like notation.

Gcom - the set of **guarded commands**, ranged over by gc

$$gc ::= b \Rightarrow c \mid gc_1 \,\square\, gc_2$$

Com - the set of **commands**, ranged over by c, ξ

$$c ::= SKIP \mid ABORT \mid x:=e \mid c_1;c_2 \mid \text{if } gc \text{ fi} \mid \text{do } gc \text{ od} \mid P?W(x) \mid P!W(e) \mid c_1 \| c_2 \mid P::c \mid \text{process } P;c$$

where $gc_1 \,\square\, gc_2$ is called an <u>alternative</u> (guarded command), if gc fi is called a <u>conditional</u> (command), do gc od is called a <u>repetitive</u> (command) and P?W(x), Q!W(e) are called <u>input</u> and <u>output</u> commands respectively. Finally, P::c denotes a process

named P with body c and process P;c is the declaration of the label P so that a process named P within c cannot be directly referenced since the scope of its name is the command c.

3.1.2. Static Semantics

Before giving a formal operational semantics for CSP, it is necessary to guarantee that all syntactic clauses considered are **valid** . This notion of syntactic validity is a device intended to capture some of the context-sensitive aspects of programming language grammars, dealing with such things as non-interference of processes. In order to capture a notion of non-interference between processes accessing a shared state we need to introduce and define the following sets.

Definition 3.1

Syn - the union of Gcom and Com, ranged over by ω.

RV(ω) - the set of variables in ω that may be read.

WV(ω) - the set of variables in ω that may be written to.

FPL(ω) - the set of free (agent) process labels contained in ω.

For a guarded command gc we also need the following predicate when giving the dynamic semantics; we give its definition here for convenience.

Bool(gc) - the disjunction of the boolean guards occurring in gc.

For each piece of syntax ω we define RV(ω), WV(ω), and FPL(ω) in a tabular fashion, the tables are to be read by cross-indexing the function required (on the vertical axis) with its argument (on the horizontal axis); thus eg. WV(P?W(x))=$\{x\}$ from the intersection of

second row (WV) and first column (P?W(x)) in the third table.

	b => c	gc_1 □ gc_2
RV	FV(b)+RV(c)	$RV(gc_1)+RV(gc_2)$
WV	WV(c)	$WV(gc_1)+WV(gc_2)$
FPL	FPL(c)	$FPL(gc_1)+FPL(gc_2)$
Bool	b	$Bool(gc_1)vBool(gc_2)$

	SKIP	ABORT	x:=e	$c_1;c_2$	if gc fi	do gc od
RV	ϕ	ϕ	FV(e)	$RV(c_1)+RV(c_2)$	RV(gc)	RV(gc)
WV	ϕ	ϕ	{x}	$WV(c_1)+WV(c_2)$	WV(gc)	WV(gc)
FPL	ϕ	ϕ	ϕ	$FPL(c_1)+FPL(c_2)$	FPL(gc)	FPL(gc)

	P?W(x)	P!W(e)	$c_1 \| \| c_2$	R::c	process R;c
RV	ϕ	FV(e)	$RV(c_1)+RV(c_2)$	RV(c)	RV(c)
WV	{x}	ϕ	$WV(c_1)+WV(c_2)$	WV(c)	WV(c)
FPL	ϕ	ϕ.	$FPL(c_1)+FPL(c_2)$	{R}	FPL(c)-{R}

Finally, the set of free variables contained in a command or a guarded command is defined by:

FV(c) = RV(c) + WV(c),

FV(gc) = RV(gc) + WV(gc).

□

The property of being a valid syntactic form is written as $\vdash$ and may be written in subscripted form such as $\vdash_{Gcom}$ gc to assert that gc is valid as a guarded command ,and $\vdash_{Com}$ c to assert that c is a valid command.

Now we can define what we mean by non-interference through shared variables in parallel processes. The idea is that each command in a parallel construct should act as if it had the (shared) state completely to itself; this makes it possible to reason about the behaviour of a parallel construct $c_1 || c_2$ in terms of the behaviour of its constituents c_1 and c_2 each alone. Consider, if c_2 can alter the value of a variable we do not want c_1 to be able to 'see' this change (by reading the value of this variable from the store) since then c_1 would be interfered with. Thus we require

$$RV(c_1) \wedge WV(c_2) = \phi$$

Similarly we also require

$$WV(c_1) \wedge WV(c_2) = \phi$$

since otherwise c_2 might interfere with the final result state produced by c_1. Thus we will require

$$FV(c_1) \wedge WV(c_2) = \phi$$

and symmetrically

$$FV(c_2) \wedge WV(c_1) = \phi$$

Definition 3.2

Let $c_1, c_2 \in Com$, then

separated(c_1, c_2) if $FV(c_1) \wedge WV(c_2) = FV(c_2) \wedge WV(c_1) = \phi$

□

The rules for deriving syntactic validity are as follows:

Guarded Commands

1. $$\frac{\vdash_{Com} c}{\vdash_{Gcom} b \Rightarrow c}$$

2. $$\frac{\vdash_{Gcom} gc_1 \quad , \quad \vdash_{Gcom} gc_2}{\vdash_{Gcom} gc_1 \,\square\, gc_2}$$

Commands

1. $\vdash_{Com}$ SKIP

2. $\vdash_{Com}$ ABORT

3. $\vdash_{Com}$ x:=e

4. $$\frac{\vdash_{Com} c_1 \ , \vdash_{Com} c_2}{\vdash_{Com} c_1;c_2}$$

5. $$\frac{\vdash_{Gcom} gc}{\vdash_{Com} \text{if } gc \text{ fi}}$$

6. $$\frac{\vdash_{Gcom} gc}{\vdash_{Com} \text{do } gc \text{ od}}$$

7. $\vdash_{Com}$ P?W(x)

8. $\vdash_{Com}$ P!W(e)

9. $$\frac{\vdash_{Com} c, \ FPL(c)=\phi}{\vdash_{Com} R::c}$$

10. $$\frac{\vdash_{Com} c}{\vdash_{Com} \text{process } R;c}$$

11. $$\frac{\vdash_{Com} c_1 \ , \vdash_{Com} c_2 \ , \text{separated}(c_1,c_2), FPL(c_1) \wedge FPL(c_2)=\phi}{\vdash_{Com} c_1 || c_2}$$

Requirements of non-interference and disjoint process names are imposed in rule 11.

The condition $FPL(c)=\phi$ in rule 9 demands that subprocesses of a named process must be declared local to that process, so for example

```
process R; process Q; process P;
    R::(P::SKIP || Q::SKIP)
```

is not a syntactically valid form whereas

```
process R;
    R::(process P; process Q;
        P::SKIP || Q::SKIP)
```

is a valid form.

3.1.3. Dynamic semantics

Having defined the syntax of CSP and its static semantics, thereby defining a language of CSP programs, we now give the dynamic semantics for commands through their associated transition system.

Definition 3.3

$$Alab = \{(N,P)W?v:\ N \in Plab+\{*\},\ P \in Plab,\ W \in Pten,\ v \in Val\} +$$
$$\{(N,P)W!v:\ N \in Plab+\{*\},\ P \in Plab,\ W \in Pten,\ v \in Val\} + \{\varepsilon\}$$

the set of action labels, ranged over by a and α, where * is introduced as standing for an unknown process name. The intended meaning of these labels is as follows:

ε	- represents a 'silent' internal action.
(N,P)W!v	- the agent named N outputs value v together with pattern W to the agent named P. Patterns allow both senders and receivers to select only communications with that pattern.
(N,P)W?v	- the agent named N receives value v from the agent named P, both using pattern W.

Note in each case N is the initiator and source of the action.

□

The <u>complement</u> of an action label a, written $\bar{a}$, is the action label describing the 'other half' of a handshake communication

involving a.

Definition 3.4

$$\bar{a} = \begin{cases} (P,Q)W?v & \text{if } a=(Q,P)W!v \\ (P,Q)W!v & \text{if } a=(Q,P)W?v \end{cases}$$

□

Definition 3.5

CSP is the transition system defined by:

config(CSP) = $\{\langle c,s\rangle : \vdash_{Com} c\} + \{\langle done,s\rangle, \langle abort,s\rangle\}$,

ranged over by y. We also allow t to range over $\{c: \vdash_{Com} c\} + \{done,abort\}$. The configuration <done,s> corresponds to successful termination with state s, while <abortion,s> corresponds to unsuccessful termination through some run-time error (Li did not give a state component to the abortion configuration, using '<u>abortion</u>' instead, but argued for this new form with the state component [Li 83 page 236]).

labels(CSP) = Alab

trans(CSP) = =>

derived from the transition relation -> ,to be defined below, by

$$\Longrightarrow = \xrightarrow{\varepsilon}{}^{*}$$

$$\xRightarrow{a} = \Longrightarrow \xrightarrow{a} \Longrightarrow \qquad \text{if } a \neq \varepsilon$$

so ε is regarded as 'silent' action.

term(CSP) = ϕ

since $y \Longrightarrow y$ for every y ε config(CSP). However, we would like to distinguish <done,s>,<abort,s> as somehow different so we introduce the set <u>aterm</u> of 'almost terminals':

aterm = $\{\langle done,s\rangle, \langle abort,s\rangle\}$

which we will use as a predicate. Note aterm(y) implies $y \xrightarrow{a}$ for any a ε Alab.

□

In defining the transition relation -> of CSP we will require an auxilliary transition relation

$$\rightarrow_{gc} \subseteq \{\langle gc,s\rangle : \vdash_{Gcom} gc\} \times labels(CSP) \times config(CSP)$$

which will be given first. The following notation is introduced for brevity;

Notation 3.1

$$\frac{r \xrightarrow{a} r_1 \mid r_2 \mid \ldots \mid r_n}{r' \xrightarrow{a'} r_1' \mid r_2' \mid \ldots \mid r_n'}$$

is an abbreviation for the n rules:

$$\frac{r \xrightarrow{a} r_i}{r' \xrightarrow{a'} r_i'} \qquad \text{where } i = 1,\ldots,n$$

□

Before we give the transition rules we briefly indicate the differences with Li's rules [Li 83 pages 67-73].

1. Errors occuring during evaluation of expressions are disallowed, this is introduced by Li in [Li 83 page 231] when proving correctness. The primary reason for this simplification is that CCS does not consider such a possibility in the semantics of the CCS construct 'if b then p else q'.

2. The configurations s and <u>abortion</u> employed by Li are replaced by <done,s> and <abort,s> respectively for reasons explained

on page 236 of Li's thesis; essentially that it makes the proof easier for the parallel construct by demanding (via the transition rules) that unsuccessful termination has no deleterious effect on the state. In Li's theory it would also make the translation easier, obviating the need for a separate process to detect abortion and reset the memory (see process R [Li 83 page 235]).

3. In addition to the replacement above we will frequently replace occurrences of s and <u>abortion</u> by <SKIP,s> and <ABORT,s> respectively. This will give us the important useful property that for any action a,

$$y \xrightarrow{a} y' \text{ , aterm}(y') \text{ implies } y \in \{\langle SKIP,s\rangle,\langle ABORT,s\rangle\}$$

Further, we change one of Li's rules for the repetitive construct from

$$\frac{\langle gc,s\rangle \xrightarrow{a}_{gc} \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle do\ gc\ od,s\rangle \xrightarrow{a} \langle do\ gc\ od,s'\rangle \mid \langle ABORT,s'\rangle}$$

to

$$\frac{\langle gc,s\rangle \xrightarrow{a}_{gc} \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle do\ gc\ od,s\rangle \xrightarrow{a} \langle SKIP;do\ gc\ od,s'\rangle \mid \langle ABORT;do\ gc\ od,s'\rangle}$$

merely to reduce the number of terms necessary in the bisimulation. This is acceptable since it only introduces more ε moves into the computations.

4. Finally, we modify the rules for parallel composition to the form used by Li for his translation [Li 83 page 231].

To sum up the changes:

Changes 1 and 4 were made by Li in chapter 6 to make

the translation work.

Change 2 was proposed by Li.

Change 3 is new but virtually trivial, just providing a useful property.

Li made one other change in chapter 6 [Li 83 section 6.2.2], restricting the allowable syntactic forms of guarded commands. The main improvement of our translation is that we show how this restriction may be dropped.

Guarded Commands

guards

$$\frac{eval(b,s)=tt \ , \ \langle c,s\rangle \xrightarrow{a} y}{\langle b \Rightarrow c,s\rangle \xrightarrow{a}_{gc} y}$$

alternative

$$\frac{\langle gc_i,s\rangle \xrightarrow{a}_{gc} \langle c,s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle \ , \ i\varepsilon\{1,2\}}{\langle gc_1 \square gc_2,s\rangle \xrightarrow{a}_{gc} \langle c,s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle}$$

Commands

assign

$$\frac{eval(e,s)=v}{\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle SKIP,s[v/x]\rangle}$$

SKIP

$$\langle SKIP,s\rangle \xrightarrow{\varepsilon} \langle done,s\rangle$$

ABORT

$$\langle ABORT,s\rangle \xrightarrow{\varepsilon} \langle abort,s\rangle$$

input

$$\langle P?W(x),s\rangle \xrightarrow{[*,P]W?v} \langle SKIP,s[v/x]\rangle$$

output

$$\frac{eval(e,s)=v}{\langle P!W(e),s\rangle \xrightarrow{[*,P]W!v} \langle SKIP,s\rangle}$$

composition

$$\frac{\langle c_1,s\rangle \xrightarrow{a} \langle c_1',s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle c_1;c_2,s\rangle \xrightarrow{a} \langle c_1';c_2,s'\rangle \mid \langle c_2,s'\rangle \mid \langle ABORT,s'\rangle}$$

conditional

1. $$\frac{\langle gc,s\rangle \xrightarrow{a}_{gc} \langle c,s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle if\ gc\ fi,s\rangle \xrightarrow{a} \langle c,s'\rangle \mid \langle SKIP,s'\rangle \mid \langle ABORT,s'\rangle}$$

2. $$\frac{eval(bool(gc),s)=ff}{\langle if\ gc\ fi,s\rangle \xrightarrow{\varepsilon} \langle ABORT,s\rangle}$$

repetition

1. $$\frac{\langle gc,s\rangle \xrightarrow{a}_{gc} \langle c,s'\rangle}{\langle do\ gc\ od,s\rangle \xrightarrow{a} \langle c;do\ gc\ od,s'\rangle}$$

2. $$\frac{\langle gc,s\rangle \xrightarrow{a}_{gc} \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle do\ gc\ od,s\rangle \xrightarrow{a} \langle SKIP;do\ gc\ od,s'\rangle \mid \langle ABORT;do\ gc\ od,s'\rangle}$$

3. $$\frac{eval(bool(gc),s)=ff}{\langle do\ gc\ od,s\rangle \xrightarrow{\varepsilon} \langle SKIP,s\rangle}$$

parallel

1. $$\frac{\langle c_1,s\rangle \xrightarrow{a} \langle c_1',s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle c_1 || c_2,s\rangle \xrightarrow{a} \langle c_1' || c_2,s'\rangle \mid \langle c_2;SKIP,s'\rangle \mid \langle c_2;ABORT,s'\rangle}$$

2. $$\frac{\langle c_2,s\rangle \xrightarrow{a} \langle c_2',s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle}{\langle c_1 || c_2,s\rangle \xrightarrow{a} \langle c_1 || c_2',s'\rangle \mid \langle c_1;SKIP,s'\rangle \mid \langle c_1;ABORT,s'\rangle}$$

3. $$\frac{\langle c_1,s\rangle \xrightarrow{a} \langle c_1',s'\rangle \ , \ \langle c_2,s\rangle \xrightarrow{a} \langle c_2',s\rangle}{\langle c_1 || c_2,s\rangle \xrightarrow{\varepsilon} \langle c_1' || c_2',s'\rangle}$$

4. $$\frac{\langle c_2,s\rangle \xrightarrow{a} \langle c_2',s'\rangle \ , \ \langle c_1,s\rangle \xrightarrow{a} \langle c_1',s\rangle}{\langle c_1 || c_2,s\rangle \xrightarrow{\varepsilon} \langle c_1' || c_2',s'\rangle}$$

Definition 3.6

$$rename_R(a) = \begin{cases} \varepsilon & \text{if } a=\varepsilon \\ (R,P)W?v & \text{if } a=(*,P)W?v,\ R \neq P \\ (R,Q)W!v & \text{if } a=(*,Q)W!v,\ R \neq Q \end{cases}$$

□

The intent of $rename_R$ is to specify actions emanating from some command c as actions of a process R::c. The conditions $R \neq P$ and $R \neq Q$ ensure that any process trying to communicate with itself will deadlock, see [Li 83 page 70] for details.

rename

$$\frac{\langle c,s\rangle \xrightarrow{a} \langle c',s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle \ , rename_R(a)=\alpha}{\langle R::c,s\rangle \xrightarrow{\alpha} \langle R::c',s'\rangle \mid \langle SKIP,s'\rangle \mid \langle ABORT,s'\rangle}$$

Definition 3.7

$$\text{scope}_{R,L}(a) = \begin{cases} a & \text{if } a=\varepsilon \\ a & \text{if } a=(N,P)W?v \text{ and } R \neq N,\ R \neq P \\ a & \text{if } a=(N,Q)W!v \text{ and } R \neq N,\ R \neq Q \\ (*,P)?W(v) & \text{if } a=(R,P)W?v \text{ and } P \notin L \\ (*,Q)?W(v) & \text{if } a=(R,Q)W!v \text{ and } Q \notin L \end{cases}$$

where $N \in Plab+\{*\}$

□

The function $\text{scope}_{R,L}$ is used to implement scoping rules for the process labelled R by renaming actions of subprocesses also labelled R. The motivation behind the scoping rules is quite subtle and complex, see [Li 83 page 71], but for our purposes in investigating translation correctness the details are not important; their translation is virtually trivial. Note that $\text{scope}_{R,L}(a)$ is a partial function.

scope

$$\frac{\langle c,s\rangle \xrightarrow{a} \langle c',s'\rangle \mid \langle done,s'\rangle \mid \langle abort,s'\rangle,\ \text{scope}_{R,FPL(c)}(a)=\alpha}{\langle process\ R;c,s\rangle \xrightarrow{\alpha} \langle process\ R;c',s'\rangle \mid \langle SKIP,s'\rangle \mid \langle ABORT,s'\rangle}$$

The following proposition will help to reduce the number of cases we need consider when constructing and verifying the bisimulation,

Proposition 3.1

$y \xrightarrow{a} y'$, aterm(y') implies $y = \langle SKIP,s\rangle$ or $y = \langle ABORT,s\rangle$

□

3.2. The translation

In this section we describe a translation from the CSP defined in

the previous section to the CCS of chapter 1. The translation is essentially that of Li given in [Li 83 chapter 6] but with one major difference: we make no restriction on the allowable forms of guarded commands. The restriction introduced by Li [Li 83 page 229] constrained guarded commands to one of 4 forms

b => SKIP;c

b => P?W(x);c

b => P!W(e);c

$gc_1 \square gc_2$

This is essentially Hoares original form of guarding ([Hoare 78]) and is usually written, with a single arrow, as (respectively)

b -> c

b,P?W(x) -> c

b,P!W(e) -> c

$gc_1 \square gc_2$

In contrast, the form introduced by Plotkin [Plotkin 82] given in the previous section has no such restriction. The merit of this form is that it is simpler and more general than the original. We will henceforth refer to Plotkins form as <u>strong guarding</u> and Hoares as <u>weak guarding</u> . The problem of translating strong guards lies in the properties of the CCS operator + with respect to 'silent' action, i.e. 1 moves in CCS. The translation of an alternative guarded command $gc_1 \square gc_2$ will be ,as Li gave it, p+q where p is the translation of gc_1 and q that of gc_2. Now, if it is the case that p, say, must read from the (simulated) store to see if gc_1 could proceed then, in performing the read, a

derivative of p will replace p+q i.e. gc_1 will appear to have been selected. However, it may be found subsequently in the computation from p that gc_1 could not in fact perform an action (if, say, it was trying to send to a non-existent receiver) and so should not have been selected since deadlock ensues. Li almost completely solved this problem by moving all the necessary read actions, for his restricted syntactic forms, to the surrounding context of the guarded command, i.e. to the conditional or repetitive command which contains it. This was achieved by defining a set GV of 'guard variables' of a guarded command whose values must be read before making a selection from the alternative guarded commands (see [Li 83 page 230]). In this section we show how GV, renamed TV for 'troublesome variables' can be extended to unrestricted guarded commands and used to guide the translation. We will make this all clear with an example when the translation has been given, but we begin with a definition of the troublesome variables of a guarded command.

3.2.1. The troublesome variables

For each command, and guarded command, we define a set of "troublesome variables" which includes all the variables whose value may need to be known in order to determine the first action(s) of the (guarded) command. This is essentially an extension of Li's set GV [Li 83 page 230] to cover all commands rather than just SKIP, P!W(e) and P?W(x). We define TV in a tabular form,

Definition 3.8

	b => c	$gc_1 \square gc_2$
TV	FV(b)+TV(c)	$TV(gc_1)+TV(gc_2)$

	SKIP	ABORT	x:=e	$c_1;c_2$	if gc fi	do gc od
TV	ϕ	ϕ	FV(e)	$TV(c_1)$	TV(gc)	TV(gc)

	P?W(x)	P!W(e)	$c_1 \| \| c_2$	R::c	process R;c
TV	ϕ	FV(e)	$TV(c_1)+TV(c_2)$	TV(c)	TV(c)

□

A few of the definitions may be unclear in the light of the (loose) prose definition of TV so we examine the tabular definition more closely:

TV(b => c) = TV(b)+TV(c): in order to proceed a guarded command must know the values of all variables in its boolean part b (to determine its truth value) and all variables necessary to determine the first move of its constituent command (see the rule guards of sec 3.1).

TV(x:=e) = FV(e): this is a useful fiction since in fact the free variables of e are not troublesome; the translation of x:=e cannot deadlock after reading them. However, using this definition allows us to reduce the number of terms in the bisimulation while still being able to prove condition 5.2 of Theorem 3.2. The alternative is to use TV(x:=e) = ϕ and add

[1.1] {(<SKIP,s[v/x]> , RS((T-X)-Y)[$store_x$!e.DONE!] with s :Y:X):
Y $\subseteq$ T-X, eval(e,s)=v }

to $R_X(x:=e)$ as defined later in section 4.1.

$\underline{TV(c_1;c_2) = TV(c_1)}$: this is because we are interested only in the first action, which must be performed by c_1.

In order to succinctly state the translation we will find it useful to define some useful shorthand descriptions of some CCS agents.

3.2.2. Auxilliary CCS definitions

To aid in the specification of the translation we introduce a number of notational identities as 'syntactic sugar' and a single agent definition, $cell_x$, for constructing a simulated state.

Simulating states: as part of the semantic translation we will need to translate the state component of a CSP configuration to a CCS term. A state is a function from Vars to Val where $Vars=\{x_n: n\in Nat\}$ is an infinite set of variables. It will be convenient to assume that all the CSP programs under consideration use variables in some initial subset $\{x_n : n \leq k\}$ of Vars for some $k \in Nat$; by making k large enough we can include any finite set of programs. CSP states will be modelled in CCS by a (finite) product of individual register cells each subscripted by the subscript number of the variable whose value they hold. Each cell can perform update and retrieval functions for its variable, so for every $n \in Nat$, define

$$cell_n(y) \Leftarrow get_n!y.cell_n(y) + store_n?v'.cell_n(v')$$

Note the actions too are subscripted by the number of the variable whose value the cell holds. Taking the parallel product of all cells with subscripts less than or equal to k, each holding the value associated with its variable in the state s, we define for each state s,

$$[\![s]\!] = \prod_{n \leq k} \mathrm{cell}_n(s(x_n))$$

This definition delivers the following desirable properties,

1. $[\![s]\!] \xrightarrow{\mathrm{get}_n!s(x_n)} [\![s]\!]$ for all $n \leq k$.
2. $[\![s]\!] \xrightarrow{\mathrm{store}_n?v} [\![s[v/x_n]]\!]$ for all $n \leq k$.
3. $[\![s]\!]$ has no transitions but 1 and 2.

To save writing a lot of subscripts we will henceforth write cell_x for cell_n if $x=x_n$, and similarly for the actions get_n and store_n.

Using simulated states: to read the values of sets of variables from the store we introduce a shorthand notation RS(X)[p] for CCS terms defined as follows for any CCS term p and X $\subseteq$ Vars,

$$RS(X)[p] = \begin{cases} p & \text{if } X=\phi \\ \mathrm{get}_n?x.RS(X-\{x\})[p] & \\ & \text{if } x=x_n \text{ and } x_m \in X-\{x\} \text{ implies } m \leq n. \end{cases}$$

The idea is that CCS and CSP share the set Vars so RS(X)[p] will read all the current values of the variables in X from the simulated state and bind them in p to those same variables, and then become this substituted version of p. This sort of process clearly needs a simulated state to interact with, so define for each CCS term p and state s

$$p \text{ with } s = (p \,|\, [\![s]\!]) \backslash (\text{Stateact} + \{\text{DONE}, \text{ABORT}\})$$

where $\text{Stateact} = \{get_x, store_x : x \in \text{Vars}\}$. Thus in 'p with s', p has exclusive access to the simulation of s and cannot signal termination (it will be part of the function of the non-behavioural relation to detect termination in other ways). As a notational convenience define for $X \subseteq \text{Vars}$

$$p \text{ with } s : X = p[\text{sub}(X,s)] \text{ with } s$$

where sub(X,s) is, as defined in chapter 1 section 2.1, the substitution of s(x) for x, if $x \in X$. Thus for example

$$RS(\{x\})[p] \text{ with } s : \phi \xrightarrow{1} p \text{ with } s : \{x\}$$

Detecting termination: in the translation, successful or abortive termination will be signalled by the actions DONE! and ABORT! respectively. Since the translation is recursive and syntax directed we will need to compose the recursive subtranslations in a manner which takes account of this convention. Firstly we will need to guarantee exclusive access to these signals from the sub-translations by having some process q exclusively monitor the sub-pieces for occurrences of these actions; so define "p monitored by q" by

$$p \wr q = (p[done,abort/DONE,ABORT] \,|\, q) \backslash \{done,abort\}$$

To model a sequential composition $c_1;c_2$ we would like q to monitor the translation of c_1 for termination; if it terminates successfully q should begin execution of the translation of c_2, otherwise abort. As a syntactic sugaring we will write

$$a \rightarrow p \qquad \text{for} \qquad a.p$$

So define

$$p_1 \text{ bef } p_2 = p_1 \{ (\text{ done? } \rightarrow p_2 \;+\; \text{abort? } \rightarrow \text{ABORT!})$$

For the more complex parallel case 'par' performs a similar function,

$$p_1 \text{ par } p_2 = p_1 | p_2 \{ (\text{ done? } \rightarrow (\text{ done? } \rightarrow \text{DONE!} \;+\; \text{abort? } \rightarrow \text{ABORT!}) \;+\; \text{abort? } \rightarrow (\text{ done? } \rightarrow \text{ABORT!} \;+\; \text{abort? } \rightarrow \text{ABORT!})$$

When only one of p_1 and p_2 has terminated we get an intermediate configuration from 'p_1 par p_2'; 'parD' if the termination was successful and 'parA' otherwise:

$$p_1 \text{ parD } p_2 = p_1 | p_2 \{ (\text{ done? } \rightarrow \text{DONE!} \;+\; \text{abort? } \rightarrow \text{ABORT!})$$

$$p_1 \text{ parA } p_2 = p_1 | p_2 \{ (\text{ done? } \rightarrow \text{ABORT!} \;+\; \text{abort? } \rightarrow \text{ABORT!})$$

At first glance it might seem that a simpler definition of 'p_1 parA p_2' might be

$$p_1 \text{ parA } p_2 = p_1 | p_2 \{ \text{ABORT!}$$

but this would not accord with the definition of CSP parallel composition wherein termination of the whole depends on the termination of <u>both</u> components; this latter definition would allow unsuccessful termination whenever just one component aborted.

3.2.3. The translation

We are now in a position to give the (syntactic) translation of commands, but first a few words on the use of subscripts and troublesome variables. The translation function for commands and

guarded commands is subscripted by an arbitrary set X of variables. It is assumed that if the set X is non-empty then the translation function is being employed to translate a sub-piece of some larger construct wherein, during execution, the surrounding context of this sub-piece will bind values for these variables before attempting its execution. Consider the translation of the output command:

$$[\![P!W(e)]\!]_X = RS(FV(e)-X)[(*,P)W!e.DONE!]$$

The use of 'FV(e)-X' reflects that it will not be necessary to bind values to the free variables of e occurring in X since they can be assumed already bound with the correct values. The subscript sets X are determined, recursively, during the translation process in the clauses for conditional and repetitive commands. Essentially, in translating, say, 'if gc fi' the idea is to make sure all the troublesome variables of gc will be bound before execution reaches the translation of gc, then translate gc under that assumption. This will ensure, by the definition of troublesome variables, no communications with the simulated store will be necessary to determine the first action of gc. We return to reinforce this point in an example after the translation has been given.

Definition 3.9

Suppose X $\subseteq$ Vars, then

$$[\![.]\!]_X : Syn \rightarrow config(L[CCS])$$

are defined as follows; (We write $[\![.]\!]$ for $[\![.]\!]_\phi$)

GUARDED COMMANDS

$[\![b \Rightarrow c]\!]_X$ = if b then $[\![c]\!]_X$

$[\![gc_1 \square gc_2]\!]_X$ = $[\![gc_1]\!]_X$ + $[\![gc_2]\!]_X$

COMMANDS

$[\![SKIP]\!]_X$ = 1.DONE!

$[\![ABORT]\!]_X$ = 1.ABORT!

$[\![x:=e]\!]_X$ = RS(FV(e)-X)[$store_x$!e.DONE!]

$[\![c_1;c_2]\!]_X$ = $[\![c_1]\!]_X$ bef $[\![c_2]\!]$

$[\![P?W(x)]\!]_X$ = (*,P)W?x.$store_x$!x.DONE!

$[\![P!W(e)]\!]_X$ = RS(FV(e)-X)[(*,P)W!e.DONE!]

$[\![$if gc fi$]\!]_X$ = RS(TV(gc)-X)[if Bool(gc)
then $[\![gc]\!]_{TV(gc)+X}$
else 1.ABORT!]

To define the translation of an iterative command we introduce for each gc ε Gcom the following definition;

D_{gc} <= RS(TV(gc))[if Bool(gc)
then $[\![gc]\!]_{TV(gc)}$ bef D_{gc}
else 1.DONE!]

Then,

$[\![$do gc od$]\!]_X$ = RS(TV(gc)-X)[if Bool(gc)
then $[\![gc]\!]_{TV(gc)+X}$ bef D_{gc}
else 1.DONE!]

$[\![c_1 || c_2]\!]_X$ = $[\![c_1]\!]_X$ par $[\![c_2]\!]_X$

$[\![R::c]\!]_X$ = $[\![c]\!]_X[ren_R]$

where ren(R) is the morphism defined by

$$ren(R)(a) = \begin{cases} rename_R(a) & \text{if } a \in Alab \\ a & \text{otherwise} \end{cases}$$

$[\![process\ R;c]\!]_X = [\![c]\!]_X[sco(R,c)]$

where sco(R,c) is the morphism defined by

$$sco(R,c)(a) = \begin{cases} scope_{R,FPL(c)}(a) & \text{if } a \in Alab \\ a & \text{otherwise} \end{cases}$$

□

An important corollary of this definition is:

Corollary 3.1

$\forall\ c \in Com.\ \forall\ X \subseteq Vars.\ FV([\![c]\!]_X) \subseteq X$

Proof by structural induction on c.
□

The semantic translation follows quite simply from the syntactic translation:

Definition 3.10

$$[\![.]\!]_{sem} : config(CSP) \rightarrow config(L[CCS])$$

is defined by

$$[\![\langle c,s\rangle]\!]_{sem} = [\![c]\!] \text{ with } s$$

$$[\![\langle done,s\rangle]\!]_{sem} = DONE! \text{ with } s$$

$$[\![\langle abort,s\rangle]\!]_{sem} = ABORT! \text{ with } s$$

As usual we drop the subscript 'sem' since no confusion can arise.

□

3.2.4. An example command translated

As an example of the functioning of the subscript sets we examine the translation of a simple command 'example_c' which is

disallowed in Li's CSP:

```
example_c      = if true => if true => P!W(x) fi
                            □
                            true => SKIP
                 fi
```

The translation of example_c is then

```
RS({x})[ if   true or true
         then [[true => if true => P!W(x) fi]]{x}
              +
              [[true => SKIP]]{x}
         else 1.ABORT! ]

= RS({x})[ if   true or true
           then if   true
                then [[if true => P!W(x) fi]]{x}
                +
                if   true
                then 1.DONE!
           else 1.ABORT! ]

= RS({x})[ if   true or true
           then if   true
                then if   true
                     then [[true => P!W(x)]]{x}
                     else 1.ABORT!
                +
                if   true
                then 1.DONE!
           else 1.ABORT! ]

= RS({x})[ if   true or true
           then if   true
                then if   true
                     then if   true
                          then (*,P)W!x.DONE!
                          else 1.ABORT!
                +
                if   true
                then 1.DONE!
           else 1.ABORT! ]
```

Clearly there is some room available for optimisation here but the point can still be discerned if we consider an equivalent (in fact congruent) agent 'eq_c' derived from [[example_c]] by replacing 'if true then p else q' by 'p' and equating 'true or true' with

'true':

```
eq_c   = RS({x})[(*,P)W!x.DONE!
                +
                1.DONE!]
```

The agent that would be produced by Li's translation method, if the syntactic restrictions he imposed were lifted to allow example_c as an argument, is, writing $[\![.]\!]_{Li}$ for his translation function,

```
          if   true
          then [[ true => if true => P!W(x) fi
                  □
                  true => SKIP ]]Li
          else 1.ABORT!

        = if   true or true
          then if   true
               then [[if true => P!W(x) fi]]Li
               +
               if   true
               then [[SKIP]]Li
          else 1.ABORT!

        = if   true or true
          then if   true
               then RS({x})[if   true
                            then [[true => P!W(x)]]Li
                            else 1.ABORT!]
               +
               if   true
               then 1.DONE!
          else 1.ABORT!

        = if    true or true
          then  if   true
                then RS({x})[if   true
                             then if   true
                                  then (*,P)W!x.DONE!
                                  else 1.ABORT!]
                +
                if   true
                then 1.DONE!
           else 1.ABORT!
```

which has an equivalent (congruent) agent,

```
eq_bad  = RS({x})[(*,P)W!x.DONE!]
          +
          1.DONE!
```

Comparing eq_c and eq_bad we see that 'eq_bad with s' can, in reading the value of x, select the output commmand for execution before executing it, i.e.

$$\text{eq\_bad with } s \stackrel{\varepsilon}{\Longrightarrow} ((*,P)W!x.\text{DONE! with } s)[s(x)/x]$$

and the derivative cannot be paired in a bisimulation with any derivative of <c,s>. In contrast 'eq_c with s' reads the value of x before attempting a choice, avoiding the trap wherein reading x makes the choice.

3.3. The statement of correctness

To give a simple statement of correctness we first identify a class of contexts, which help to characterise agents which have run their course. We begin by considering the following question: if

$$[\![<c,s>]\!] \stackrel{w}{\Longrightarrow} p \text{ with } \$ \not\rightarrow$$

for some $w \in Alab^*$, what are the forms p may take given c (but not s)? Intuitively, for each c we want to know when 'p with s' has terminated (rather than deadlocked) and in what way. To do this we define two sets for each (guarded) command ω,

$tf(\omega)$	the terminated forms of the translation of c.
$atf(\omega)$	the almost-terminated forms of the translation of ω; they are willing to signal termination but have not yet done so. These sets are defined from consideration of the translation;

Definition 3.11

	atf	tf
b => c	atf(c)	tf(c)
gc_1 □ gc_2	$tf(gc_1)+tf(gc_2)$	$atf(gc_1)+atf(gc_2)$
SKIP	{DONE!}	{Nil}
ABORT	{ABORT!}	{Nil}
x:=e	{DONE!}	{Nil}
P?W(x)	{DONE!}	{Nil}
P!W(e)	{DONE!}	{Nil}
if gc fi	atf(gc) + {ABORT!}	tf(gc) + {Nil}
$c_1;c_2$	{p_1⟨p_2, p_1⟨Nil: $p_1 \varepsilon tf(c_1)$, $p_2 \varepsilon atf(c_2)$}	{p_1⟨p_2, p_1⟨ABORT!: $p_1 \varepsilon tf(c_1)$, $p_2 \varepsilon tf(c_2)$ }
do gc od	{p⟨q, p⟨ABORT!,DONE!: $p \varepsilon tf(gc)$, $q \varepsilon atf(do\ gc\ od)$ }	{p⟨q, p⟨Nil, Nil: $p \varepsilon tf(gc)$, $q \varepsilon tf(do\ gc\ od)$ }
$c_1 \| \| c_2$	{p|q⟨DONE!, p|q⟨ABORT!: $p \varepsilon tf(c_1)$, $q \varepsilon tf(c_2)$}	{p|q⟨Nil: $p \varepsilon tf(c_1)$, $q \varepsilon tf(c_2)$ }
R::c	{p[Ren(R)]: $p \varepsilon atf(c)$ }	{p[Ren(R)]: $p \varepsilon tf(c)$ }
process R;c	{p[sco(R,c)]: $p \varepsilon atf(c)$ }	{p[sco(R,c)]: $p \varepsilon tf(c)$ }

□

There are two points to note, firstly there are no action operators in $tf(\omega)$ for any ω, and secondly for each $p \in atf(\omega)$ there is exactly one action label, either DONE! or ABORT!. Thus we can define a function 'lastact' from almost-terminated forms, delivering this action label and we can introduce predicates for recognising terminal and almost-terminal forms.

Definition 3.12

$null(p)$ if $\nexists\, \omega.\ p \in tf(\omega)$

$nuld(p)$ if $\nexists\, \omega.\ p \in atf(\omega)$, $lastact(p) = DONE!$

$nula(p)$ if $\nexists\, \omega.\ p \in atf(\omega)$, $lastact(p) = ABORT!$

□

The following proposition supports the intent of the previous definitions.

Proposition 3.2

1. $nuld(p)$ implies

   a) There is a unique $q.\ p \xrightarrow{DONE!} q$

   b) $p \xrightarrow{a}$ implies $a = DONE!$

2. $nula(p)$ implies

   a) There is a unique $q.\ p \xrightarrow{ABORT!} q$

   b) $p \xrightarrow{a}$ implies $a = ABORT!$

□

We are now in a position to state the relation 'result' which constitutes the non-behavioural component of the correctness

criteria. Basically it demands that if an almost-terminal configuration of CSP is paired with some CCS agent then that agent is of the form 'p with s' where s is the state component of the CSP configuration. Further, p has only a single action left; DONE! if the CSP configuration is a successful termination, ABORT! if it is unsuccessful. Conversely, any pairing (y,p with s) such that p is willing to signal termination (by DONE! or ABORT!) must guarantee that y is a configuration very close to the appropriate termination (with the same state) and p is 'dead' after its signal.

For simplicity we make no demands in result for deadlocked CSP configurations, nor demands about convergence,though the translation would support interesting results for these notions.

Definition 3.13

Define **result** $\underline{C}$ config(CSP) x config(L[CCS]) by

y result p with $

iff

1. y = <done,s> implies s=$, nuld(p)
2. y = <abort,s> implies s=$, nula(p)
3. $p \xrightarrow{\underline{DONE!}}$ implies $y \in \{<done,\$>,<SKIP,\$>\}$, nuld(p)
4. $p \xrightarrow{\underline{ABORT!}}$ implies $y \in \{<abort,\$>,<ABORT,\$>\}$, nula(p)

□

Theorem 3.1

The translation [[.]]: config(CSP) -> config(L[CCS]) is result-good.

Proof: The proof is the subject of the next section.

□

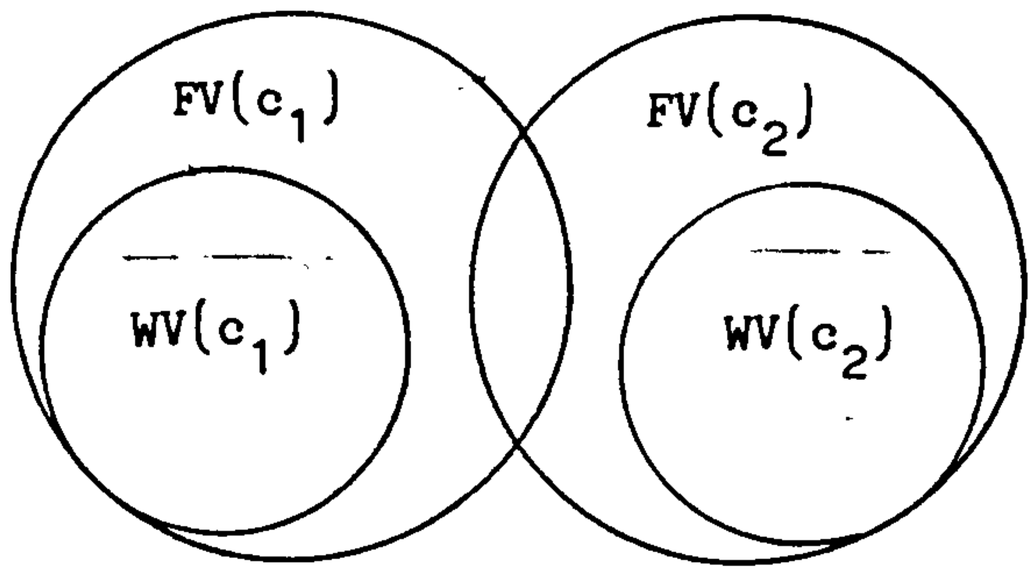
3.4. The proof of correctness

The purpose of this section is to construct a result-bisimulation $R \subseteq config(CSP) \times config(L[CCS])$ such that $y\ R\ [\![y]\!]$ for every $y \in config(CSP)$. The construction will proceed by structural induction on the possible forms of CSP commands: the bisimulation we construct for configurations of the form $y = \langle c_1;c_2,s\rangle$, say, will assume constructed bisimulations for the configurations of the form $\langle c_1,s\rangle$ and $\langle c_2,s\rangle$. The required bisimulation R is then the union of the bisimulations for each command c.

Before presenting the bisimulations we need a definition necessary for synthesising and analysing computations of $c_1||c_2$ in terms of computations of c_1 and computations of c_2. The notion of the merging of states, given here as a partial function *, is intimately connected with the notion of separation of processes. Reviewing Definition 3.2 we recall two commands c_1,c_2 altering a common store, for each c_i $(i \in \{1,2\})$ the set of variables whose value may be altered was $WV(c_i)$, those referred to at all was $FV(c_i)$, and the definition of separation stated

$$WV(c_i) \wedge FV(c_j) = \phi \quad ,\{i,j\}=\{1,2\}.$$

This can be represented pictorially with a Venn diagram of the set Vars,



Now suppose c_1, c_2 proceed in parallel, altering a common initial state 'init_s' to $\$_1$ and $\$_2$ respectively. Then the changes made by c_1 in $\$_1$ are invisible to c_2, and vice versa. We can now define for such states $\$_1, \$_2$ a composite state s which is the same as $\$_1$ and $\$_2$ where they both agree (i.e. on Vars-WV(c_1)-WV(c_2)) but like $\$_1$ on WV(c_1) and $\$_2$ on WV(c_2). Thus s collects the changes made in $\$_1$ and $\$_2$ and is well-defined since the changes are on disjoint sets.

Definition 3.14

Suppose $W_i \subseteq$ Vars i=1,2,

$W_1 \wedge W_2 = \phi$,

$s_1(x)=s_2(x)$ if $x \notin W_1+W_2$.

Define

*: States x States -> States

$$s_1 * s_2(x) = \begin{cases} s_i(x) & \text{if } x \in W_i \\ s_1(x) & \text{otherwise} \end{cases}$$

A notational point: in all instances of its use at least one argument will be subscripted to disambiguate s_1, s_2. For example,

$\$*s_j$

means $\$*s_2$ if j=2

or $s_1*\$$ if j=1

i.e. the argument order is reversed if j=1. Further, * should more accurately be written $*_{W1,W2}$, but in all uses of * W_1 and W_2 will be defined in the immediate context (In fact only one use of * is made in the bisimulation, and therein W_1 and W_2 are the written variables of c_1 and c_2 respectively in a parallel command $c_1||c_2$).

□

To extract the constituent commands of a guarded command we define a set 'commands(gc)',

Definition 3.15

	commands
b => c	{c}
$gc_1 \square gc_2$	commands(gc_1)+commands(gc_2)

□

Before we can give R_χ(do gc od) we need to define the set of processes resulting from the recursive execution of the translation of 'do gc od'. The idea is that during any simulated iteration of gc in do gc od there may be extant terms from previous iterations. We wish to concentrate on the currently active term 'p', say, so we define for each CCS term p the set 'ITER(p)' of possible forms for p with its environment of debris from previous iterations.

Definition 3.16

Let p be any CCS term and gc ε Gcom, then define

$$ITER(p) = \{ r_1 \triangleright (r_2 \triangleright (\ldots r_n \triangleright p)\ldots): null(r_i)\ i=1,..,n,\ n\geq 1\}$$

and let iter(p) range over ITER(p).

□

Relating this definition to that of terminated forms we see,

Corollary 3.2

$\forall$ gc ε Gcom. tf(do gc od) $\underline{C}$ ITER(Nil)

□

3.4.1. The bisimulation

We now define a bisimulation $R_X(c)$ for every X $\underline{C}$ Vars and c ε Com, the union over X and c of these sets will be the required result-bisimulation. As a notational convenience we allow $R_X(S)$ where S is a subset of Com to stand for the union over c ε S of $R_X(c)$. Also we allow R(c) to stand for $R_\phi(c)$. Note in each set of pairs numbered [n], for some n, there is an implicit universal quantification over unbound variables.

Definition 3.17

END is the following set of pairs;

$$\{(\langle SKIP,s\rangle \text{ , p with s}), (\langle done,s\rangle \text{ , p with s}): \text{nuld(p)}\}$$

$$+ \quad \{(\langle ABORT,s\rangle \text{ , p with s}), (\langle abort,s\rangle \text{ , p with s}): \text{nula(p)}\}$$

□

Definition 3.18

R_X(SKIP)

[1] {(<SKIP,s> , 1.DONE! with s)}

[2] END

R_X(ABORT)

[1] {(<ABORT,s> , 1.ABORT! with s)}

[2] END

R_X(x:=e)

[1] {(<x:=e,s> , RS((FV(e)-X)-Y)[$store_x$!e.DONE!] with s :Y:X):
Y $\underline{C}$ FV(e)-X }

[2] END

$R_X(c_1;c_2)$

[1] {(<ξ;c_2,s> , p bef $[\![c_2]\!]$ with $):
<ξ,s> $R_X(c_1)$ p with $ }

[2] {(<ξ,s> , p ξ q with $):
null(p),
<ξ,s> $R(c_2)$ q with $ }

[3] END

R_X(P?W(x))

[1] {(<P?W(x),s> , (*,P)W?x.$store_x$!x.DONE! with s)}

[2] {(<SKIP,s[v/x], $store_x$!v.DONE! with s)}

[3] END

R_X(P!W(e))

[1] {(<P!W(e),s> , RS((FV(e)-X)-Y)[(*,P)W!e.DONE!] with s :Y:X):
Y $\subseteq$ FV(e)-X }

[2] END

R_X(if gc fi) Let T=TV(gc);

[1] {(<if gc fi,s> , RS((T-X)-Y)[if Bool(gc)
then $[\![gc]\!]_{T+X}$
else 1.ABORT!] with s :Y :X):
Y $\subseteq$ T-X }

[2] R_{T+X}(commands(gc))

[3] END

R_X(do gc od) Let T=TV(gc),

Here the difference between the two pair forms [1.1] and [1.2] arises because on the first iteration (only) we can assume values will be bound for variables in X.

[1.1] {(<do gc od,s> , RS((T-X)-Y)[if Bool(gc)
then $[\![gc]\!]_{T+X}$ bef D_{gc}
else 1.DONE!]
with s :Y :X):
Y $\underline{C}$ T-X }

[1.2] {(<do gc od,s> , iter(D_{gc}) with s),
(<do gc od,s> , iter(RS(T-Z)[if Bool(gc)
then $[\![gc]\!]_{T}$ bef D_{gc}
else 1.DONE!])
with s : Z):
Z $\underline{C}$ T }

[2.1] {(<ξ;do gc od,s> , p bef D_{gc} with $):
<ξ,s> R_{T+X}(commands(gc)) p with $ }

[2.2] {(<ξ;do gc od,s> , iter(p bef D_{gc}) with $):
<ξ,s> R_{T}(commands(gc)) p with $ }

[3] END

$R_X(c_1||c_2)$ Let * = $*_{W1,W2}$'

[1] {(<$\xi_1||\xi_2$,s> , p_1 par p_2 with $):
<ξ_i,s_i> $R_X(c_i)$ p_i with $$_i$,i=1,2
s_1*s_2=s, $\$_1*\$_2$=$ }

[2] {(<ξ;SKIP,s> , p_1 parD p_2 with $):
null(p_i),
<ξ,s_j> $R_X(c_j)$ p_j with $$_j$,
{i,j}={1,2},
$s*s_j$=s, $s*\$_j$=$ }

[3] {(<ξ;ABORT,s> , p_1 parA p_2 with $): null($p_i$),
<ξ,s_j> $R_X(c_j)$ p_j with $$_j$,
{i,j}={1,2},
s*s_j=s, s*$$_j$=$ }

[4] END

R_X(process R;c)

[1] {(<process P;ξ,s> , p[sco(P,c)] with $): ξ with s R_X(c) p with $ }

[2] END

R_X(P::c)

[1] {(<P::ξ,s> , p[ren(P)] with $): <ξ,s> R_X(c) p with $ }

[2] END

□

Before we attempt a proof that the sets presented above constitute a bisimulation we note a few simple properties of those sets which will be useful later on, beginning with some definitions. Firstly, in implementing CSP in CCS we have required extra action labels for communication with the simulated store, detection of termination and silent CCS action. Clearly we could add these actions, and others if need be, to labels(CSP) so that labels(CSP) = labels(L[CCS]) as the simple theory required, but its only a minor point so we won't bother.

Definition 3.19

The CCS action labels we have employed in translating CSP are given by

$$\begin{aligned} \text{Actimp} = \; & \{\text{store}_x, \text{get}_x : x \in \text{Vars}\} + \\ & \{1, \text{DONE}, \text{ABORT}, \text{done}, \text{abort}\} + \\ & \{(N,P)W : N \in \text{Plab} + \{*\},\ P \in \text{Plab},\ W \in \text{Pten}\} \end{aligned}$$

□

It will be useful to extend the notions of free, written, and read variables, as given in Definition 3.1 for CSP commands, to their translations (and thereby to their derivatives) by consideration of the translation, wherein for any X,

$$\begin{aligned} \text{get}_x \in \text{sort}([\![c]\!]_X) \quad & \text{implies } x \in RV(c) \\ \text{store}_x \in \text{sort}([\![c]\!]_X) \quad & \text{implies } x \in WV(c) \end{aligned}$$

Definition 3.20

Suppose $\text{sort}(p) \subseteq \text{Actimp}$, define the sets of (simulated) written, read, and free variables by

$$\begin{aligned} swv(p) &= \{x : \text{store}_x \in \text{sort}(p)\} \\ srv(p) &= \{x : \text{get}_x \in \text{sort}(p)\} \\ sfv(p) &= swv(p) + srv(p) \end{aligned}$$

□

Now Definition 3.2 can be accordingly extended with a useful corollary,

Definition 3.21

Suppose p,q are CCS terms, sort(p) $\underline{C}$ Actimp, sort(q) $\underline{C}$ Actimp,

then

separated(p,q) iff swv(p) $\wedge$ sfv(q) = swv(q) $\wedge$ sfv(p) = ϕ

□

Corollary 3.3

Let $c_1, c_2 \varepsilon$ Com, then for any X $\underline{C}$ Vars

separated(c_1, c_2) implies separated($[\![c_1]\!]_X, [\![c_2]\!]_X$)

□

The following proposition states some useful properties of $R_X(c)$

for any c ε Com,

Proposition 3.3

Suppose X $\underline{C}$ Vars, c ε Com, then

1. END $\underline{C}$ $R_X(c)$
2. Suppose for some c ε Com we have y $R_X(c)$ p with s, then
   1. FV(p) $\underline{C}$ X
   2. sort(p) $\underline{C}$ Actimp

Proof by structural induction on c.

□

3.4.2. The proof

In this subsection we prove a single theorem, the upshot of which is that R(c) is a result-bisimulation containing the pair (<c,s>, $[\![$<c,s>$]\!]$) for every state s. That $[\![.]\!]$: config(CSP) -> config(L[CCS]) is result-good is then immediate.

We start with a lemma establishing that the relation END, which by Proposition 3.3 is a subset of every $R_X(c)$, is a result-bisimulation in itself.

Lemma 3.1

END is a result-bisimulation.

□

Theorem 3.2

Suppose X $\underline{C}$ Vars, c ε Com, s ε States, then

1. <c,s> $R_X(c)$ $[\![c]\!]_X$ with s :X
2. $R_X(c)$ $\underline{C}$ result
3. Suppose y $R_X(c)$ p, then
   1. If p $\xrightarrow{a}$ q then either
      (a) a = 1, $\exists$ y'. y $\xrightarrow{\varepsilon}$ y' $R_X(c)$ q
      (b) a = 1, y $R_X(c)$ q
      (c) a ≠ 1, $\exists$ y'. y $\xrightarrow{a}$ y' $R_X(c)$ q
   2. If y $\xrightarrow{a}$ y' then $\exists$ q. p $\xRightarrow{a}$ q, y' $R_X(c)$ q
4. Suppose c = if gc fi or c = do gc od, X $\underline{C}$ Vars, and TV(gc) = T then
   1. $[\![gc]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r, a≠1
      implies
      $\exists$ y. <gc,s> $\xrightarrow{a}$ y R_X(commands(gc)) r
   2. $[\![gc]\!]_{T+X}$ with s :T+X $\xrightarrow{1}$ r
      implies
      $\exists$ y. <gc,s> $\xrightarrow{\varepsilon}$ y R_X(commands(gc)) r
   3. <gc,s> $\xrightarrow{a}$ y
      implies
      $\exists$ r. $[\![gc]\!]_{T+X}$ with s :T+X $\xRightarrow{a}$ r, y R_X(commands(gc)) r

5. Suppose X $\underline{C}$ Vars and TV(c)=T, then

   1. $[\![c]\!]_{T+X}$ with s :T+X $-\underline{a}\rightarrow$ r , a ≠ 1

      implies

      $\nexists$ y. <c,s> $-\underline{a}\rightarrow$ y $R_{T+X}(c)$ r

   2. $[\![c]\!]_{T+X}$ with s :T+X $-\underline{1}\rightarrow$ r

      implies

      $\nexists$ y. <c,s> $-\underline{\varepsilon}\rightarrow$ y $R_{T+X}(c)$ r

□

Before launching into the proof of this theorem it is instructive to pause and consider exactly what it says, considering each consequent in turn:

1 This will guarantee that for every configuration of CSP which is not of the form <done,s> or <abort,s> we have a bisimulation containing it paired with its translation, relative to any set X of variables, substituted by the appropriate values for variables in X. Taking X=ϕ we get <c,s> R(c) $[\![$<c,s>$]\!]$ as required by result-goodness. This part is easily proved by looking through the bisimulations for the appropriate pair.

2 This is the requirement that the bisimulation be a result-bisimulation. In establishing this condition we concentrate on the cases involving successful termination, the corresponding unsuccessful cases are proved similarly.

3 A simple inductive argument shows this condition implies the bisimulation closure condition R $\underline{C}$ F(R) for the transition relations $\Rightarrow_{CCS}$ and $\Rightarrow_{CSP}$ (i.e. trans(L[CCS]) and

trans(CSP)). We use this stronger form here simply to reduce the number of cases and definitions required in the proof. The proof will concentrate on condition 3.1 for exemplary cases since condition 3.2 is more easily accepted by simple inspection and the remaining cases introduce no new techniques or definitions.

We shall employ the abbreviated notation introduced in chapter 2 for describing the cases in the bisimulation proof, with the added feature of decorations when proving 3.1 to say which subcase a, b or c was satisfied.

4 This is where the earlier discussion of troublesome variables reaches formal fruition; these results relate the first actions of a guarded command to those of its translation and are used in proving the conditional and iterative translations correct.

5 To carry the proof of 4 through an inductive step we will need a property similar to 4 for commands since the first step, if any, of 'b => c' is the first step of c.

Proof: We proceed by structural induction on c, in each case of c we refer to the set of pairs $R_\chi(c)[n]$ as [n];

case c = SKIP :

1. [1] contains the required pairs.

2. Only pairs in END need examination, with immediate confirmation.

3. Obviously there are very few transitions to be considered;

For 3.1 :

$$[1] \xrightarrow{1} \text{END} \quad (a)$$

For 3.2 :

$[1] \xrightarrow{\varepsilon} \text{END}$

4. Clearly c is not a conditional or repetitive command.

5. TV(SKIP) = ϕ, so

$[\![\text{SKIP}]\!]_X$ with s :X $\xrightarrow{a}$ r

implies a =1 and r = DONE! with s, but

<SKIP,s> $\xrightarrow{\varepsilon}$ <done,s> END r

case c = ABORT : the proof is essentially as for SKIP.

case c = x:=e :

1. The required pairs are contained in [1].

2. Only END need be examined, with immediate confirmation.

3. The transitions may be examined as follows;

For 3.1 :

$[1] \xrightarrow{1} [1]$ (b) Reading variables

$[1] \xrightarrow{1} \text{END}$ (a) The assignment

For 3.2 :

$[1] \xrightarrow{\varepsilon} \text{END}$

4. Clearly c is neither a conditional nor an iterative command.

5. TV(x:=e) = FV(e) so

$[\![x:=e]\!]_{T+X} = \text{store}_x!e.\text{DONE}!$

thus $[\![x:=e]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r implies

a = 1, r = DONE! with s[v/x], eval(e,s) = v

Hence we establish 5.2 since

<x:=e,s> $\xrightarrow{\varepsilon}$ <SKIP,s[v/x]> END r

case c = P?W(x) :

1. Those required pairs are in [1].

2. Only END need be examined, with immediate confirmation.

3. The transitions may be analysed as follows;

For 3.1 :

[1] $-\underline{(*,P)W?v}\rightarrow$ [2] (c)

[2] $-\underline{1}\rightarrow$ END (a)

For 3.2 :

[1] $-\underline{(*,P)W?v}\rightarrow$ [2]

[2] $-\underline{\varepsilon}\rightarrow$ END

4. Clearly c is not of the supposed form.

5. $TV(P?W(x)) = \phi$ (so $T+X = X$) and

$[\![P?W(x)]\!]_X$ with s :X $-\underline{a}\rightarrow$ r

implies $\exists$ v ε Val such that

a = (*,P)W?v, r = $store_x$!v.DONE! with s :X

but we can establish 5.1 by

<P?W(x),s> $-\underline{(*,P)W?v}\rightarrow$ <SKIP,s[v/x]> $R_X(c)$[2] r

case c = P!W(e) :

1. Those pairs required are in [1].

2. Only END need be examined, with immediate confirmation.

3. The transitions may be analysed as follows;

For 3.1 :

[1] $-\underline{1}\rightarrow$ [1] (b) Reading variables

[1] $-\underline{(*,P)W!v}\rightarrow$ END (c)

For 3.2 :

[1] $-\underline{(*,P)W!v}\rightarrow$ END

4. Clearly c is neither a conditional nor an iterative command.

5. $TV(P!W(e)) = FV(e)$, so

$[\![P!W(e)]\!]_{T+X}$ = (*,P)W!e.DONE!

and we only need consider case 5.1

5.1 Suppose

$[\![P!W(e)]\!]_{T+X}$ with s :T+X $-\underline{a}\rightarrow$ r

then a = (*,P)W!v, eval(e,s) = v, r = DONE! with s

Accordingly,

<P!W(e),s> $-\underline{a}\rightarrow$ <SKIP,s> END r

case c = $c_1;c_2$:

1. The required pairs are contained in [1] because by the inductive hypothesis

$\langle c_1,s\rangle\ R_X(c_1)\ [\![c_1]\!]_X$ with s :X

so

$\langle c_1;c_2,s\rangle\ R_X(c_1;c_2)[1]\ [\![c_1]\!]_X$:X bef $[\![c_2]\!]$ with s

but

$[\![c_1]\!]_X$:X bef $[\![c_2]\!]$ with s = $[\![c_1]\!]_X$ bef $[\![c_2]\!]$ with s :X

since by Corollary 3.1

$FV([\![c_2]\!]) = \phi$

2. We proceed by cases on the pair forms;

[1]. Clearly y = <done,s>, and p bef q $-\underline{\text{DONE!}}\not\rightarrow$ for any p,q.

[2]. Clearly y = <done,s>, and p ξ q $-\underline{\text{DONE!}}\rightarrow$ implies

q $-\underline{\text{DONE!}}\rightarrow$ (since null(p)) so then ξ = SKIP

and nuld(q) (by the inductive hypothesis) and hence

also nuld(p ξ q) since null(p).

3. The transitions may be analysed as follows;

For 3.1 :

[1]	$-\underline{a}\rightarrow$ [1]	(a,b,c)	Executing command ξ.
[1]	$-\underline{1}\rightarrow$ [2]	(a)	ξ = SKIP.
[1]	$-\underline{1}\rightarrow$ END	(a)	ξ = ABORT.
[2]	$-\underline{a}\rightarrow$ [2]	(a,b,c)	Executing command ξ.
[2]	$-\underline{1}\rightarrow$ END	(a)	ξ = SKIP or ξ = ABORT.

For 3.2 :

$[1] \xrightarrow{a} [1]$

$[1] \xrightarrow{\varepsilon} [2]$

$[1] \xrightarrow{\varepsilon} \mathrm{END}$

$[2] \xrightarrow{a} [2]$

$[2] \xrightarrow{\varepsilon} \mathrm{END}$

We just take [1] as an exemplar, so suppose

p bef $[\![c_2]\!]$ with \$ $\xrightarrow{a}$ r

and let $R_1 = R_\chi(c_1)$, $R_2 = R(c_2)$.

We proceed by cases on the derivation,

1. p communicates with the translated state, i.e.

$\exists\, \alpha, p', \$'$ such that

$p \xrightarrow{\alpha} p'$, $[\![\$]\!] \xrightarrow{\bar{\alpha}} [\![\$']\!]$,

$a = 1$, $r = p'$ bef $[\![c_2]\!]$ with \$'

then

p with \$ $\xrightarrow{1}$ p' with \$'

so by the inductive hypothesis

either $\langle \xi, s \rangle \; R_1 \; p'$ with \$'

or $\exists\, y.\ \langle \xi, s \rangle \xrightarrow{\varepsilon} y \; R_1 \; p'$ with \$'

In the former case we have immediately

$\langle \xi; c_2, s \rangle \; R_\chi(c_1; c_2)[1] \; r$

(establishing (b)), and in the latter we must proceed

by cases on y;

1. y=<done,s>: then

nuld(p'), \$'=s, ξ = SKIP

from which we see

$\langle \xi, s \rangle$ END p' with \$'

so $\langle \xi, s \rangle \; R_1 \;$ p' with \$'

whence we can establish (b) since

$\langle \xi; c_2, s \rangle \; R_X(c_1; c_2)[1] \; r$

2. y=<abort,s>: is treated very like the above case.

3. $\exists$ ξ', s'. y=<ξ',s'> : then immediately

$\langle \xi; c_2, s \rangle \xrightarrow{\varepsilon} \langle \xi'; c_2, s' \rangle \; R_X(c_1; c_2)[1] \; r$

establishing (a).

2. p acts on its own, i.e. $\exists$ p' such that

$p \xrightarrow{a} p', \; r = p' \text{ bef } [\![c_2]\!] \text{ with } \$$

then

$p \text{ with } \$ \xrightarrow{a} p' \text{ with } \$$

and the proof proceeds as in case 1.

3. p terminates successfully, i.e. $\exists$ p' such that

$p \xrightarrow{\underline{\text{DONE!}}} p', \; r = p' \; \wr \; [\![c_2]\!] \text{ with } \$, \; a = 1$

then

$\xi = \text{SKIP}, \; s = \$, \; \text{null}(p')$

so we establish (a) since

$\langle \xi; c_2, s \rangle \xrightarrow{\varepsilon} \langle c_2, s \rangle \; R_X(c_1; c_2)[2] \; r$

4. p terminates unsuccessfully: this case is very similar

to that in 3.

case c = if gc fi :

1. Those required pairs are contained in [1].

2. For pairs in [2] we can apply the inductive hypothesis, for pairs in END the result is immediate; this just leaves pairs in [1]:

[1]. Clearly y = <done,s> so suppose

$$RS((T-X)-Y)[\text{if } \; \text{Bool}(gc) \; \text{then } [\![gc]\!]_{T+X} \; \text{else 1.ABORT! }] \text{ with s :T+X} \xrightarrow{\underline{\text{DONE!}}}$$

then

$[\![gc]\!]_{T+X}:T+X \xrightarrow{DONE!}$

so $\exists\ \xi\ \varepsilon$ commands(gc). $[\![\xi]\!]_{T+X}:T+X \xrightarrow{DONE!}$

hence by the inductive hypothesis (2) ξ = SKIP, but

$[\![SKIP]\!]_{T+X}:T+X \xrightarrow{DONE!}\not>$

whence our initial supposition was false.

3. The transitions may be analysed as follows;

For 3.1 :

[1] $\xrightarrow{1}$ [1]	(b)	Reading troublesome variables.
[1] $\xrightarrow{a}$ [2]	(a,c)	First action of $[\![gc]\!]$.
[1] $\xrightarrow{1}$ END	(a)	All booleans false, so abort.

For 3.2 :

[1] $\xrightarrow{a}$ [2]

[1] $\xrightarrow{\varepsilon}$ END

The only non-trivial case is pairs in [1]; suppose

$Y \subseteq T-X$, RS((T-X)-Y)[p] with s $\xrightarrow{a}$ r,

p = if Bool(gc) then $[\![gc]\!]_{T+X}$ else 1.ABORT!

We proceed by cases on the derivation

1. Reading troublesome variables, i.e. $\exists\ x \in (T-X)-Y$ such that

$Y' = Y+\{x\}$, a = 1, r = RS((T-X)-Y')[p] with s :Y':X

then immediately, verifying (b), we have

<if gc fi,s> R_X(c)[1] r

2. The boolean parts of all the guards are false and the command aborts, i.e.

Y = T-X, a = 1, eval(Bool(gc)[sub(T+X,s)]) = ff,

r = ABORT! with s

then, verifying (a), we have

eval(Bool(gc),s) = ff,

<if gc fi,s> $\xrightarrow{\varepsilon}$ <ABORT,s> END r

3. This is the first action of the translated guarded command,

   and it isn't a 1, i.e.

   $[\![gc]\!]_{T+X}$:T+X with s $\xrightarrow{a}$ r, a≠1,

   eval(Bool(gc)[sub(T+X,s)]) = tt

   then by 4.1 ⊣ y such that

   <gc,s> $\xrightarrow{a}$ y R_{T+X}(commands(gc)) r

   whence we have established (c) since

   <if gc fi,s> $\xrightarrow{a}$ y R_X(c)[2] r

4. This is the first action of the translated guarded command,

   and it is a 1; this case is very like 3 except that we invoke

   4.2 to establish (a).

4. Let T = TV(gc), then

   4.1 We proceed by structural induction on gc;

   BASE gc = b => ξ: Then let R = R_{T+X}(ξ) (so R $\subseteq$ R_X(c)[2]) and suppose

   if b then $[\![\xi]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r, a≠1,

   then by CCS semantics

   $[\![\xi]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r, eval(b[sub(T+X,s)])=tt,

   hence by the inductive hypothesis (5.1) on ξ and Definition 1.4

   ⊣ y. <ξ,s> $\xrightarrow{a}$ y R r, eval(b,s)=tt

   so by CSP semantics (guards)

   <b => ξ ,s> $\xrightarrow{a}$ y R r

   STEP gc = $gc_1 \square gc_2$: Suppose

   $[\![gc_1]\!]_{T+X}$ + $[\![gc_2]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r, a≠1,

   then for some i ε {1,2},

   $[\![gc_i]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r

   so by the inductive hypothesis on gc_i,

   ⊣ y. <gc_i,s> $\xrightarrow{a}$ y R_{T+X}(c) r

   and hence by CSP semantics (alternative)

$\langle gc_1 \square gc_2, s\rangle \xrightarrow{a} y$

4.2 and 4.3 follow similarly.

5. Let T=TV(c), then T=TV(gc) and Bool(gc)[sub(T+X,s)] is closed

5.1. Since a ≠ 1 we need only consider the case

$[\![gc]\!]_{T+X}$ with s :T+X $\xrightarrow{a}$ r

then by 4.1

$\exists$ y. $\langle gc,s\rangle \xrightarrow{a} y\ R_X(c)[2]\ r$

and by semantics of CSP (conditional 1) and Definition 1.4

$\langle$if gc fi,s$\rangle \xrightarrow{a} y$

5.2. Suppose

if Bool(gc)

then $[\![gc]\!]_{T+X}$

else 1.ABORT! with s :T+X $\xrightarrow{1}$ r

then we have two cases;

1. eval(Bool(gc)[sub(T+X,s)])=tt, $[\![gc]\!]_{T+X}$ with s :T+X $\xrightarrow{1}$ r :

By 4.2 we have

$\exists$ y. $\langle gc,s\rangle \xrightarrow{\varepsilon} y\ R_X(c)[2]\ r$

so by CSP semantics (conditional 1)

$\langle$if gc fi,s$\rangle \xrightarrow{\varepsilon} y.$

2. eval(Bool(gc)[sub(T+X,s)])=ff, r=ABORT! with s :

Apply CSP semantics, Definition 1.4 and use END.

case c = do gc od :

1. Those required pairs are in [1.1].

2. The proof here mirrors those for the conditional and sequential commands, the approach is the same.

3. The transitions may be analysed as follows;

For 3.1 :

$[1.n] \xrightarrow{1} [1.n]$ (b) Reading troublesome variables.

$[1.n] \xrightarrow{a} [2.n]$ (a,c) First action of [[gc]].

$[1.n] \xrightarrow{1} END$ (a) No booleans true.

$[2.n] \xrightarrow{a} [2.n]$ (a,b,c) Executing command ξ.

$[2.n] \xrightarrow{1} [1.2]$ (a) ξ = SKIP.

$[2.n] \xrightarrow{1} END$ (a) ξ = ABORT.

For 3.2 :

$[1.n] \xrightarrow{a} [2.n]$

$[1.n] \xrightarrow{\varepsilon} END$

$[2.n] \xrightarrow{a} [2.n]$

$[2.n] \xrightarrow{\varepsilon} [1.2]$

$[2.n] \xrightarrow{\varepsilon} END$

The proof is very like that given for the alternative construct except for the accretion of debris.

4. The proof is the same as that given for the alternative construct.

5. Let T=TV(do gc od)=TV(gc);

5.1. Suppose

$$[[gc]]_{T+X} \text{ bef } D_{gc} \text{ with } s : T+X \xrightarrow{a} r,\ a \neq 1$$

then since $a \neq 1$ $\vdash$ p such that

$$r = p \text{ bef } D_{gc} \text{ with } \$,$$

$$eval(Bool(gc)[sub(T+X,s)]) = tt,$$

$$[[gc]]_{T+X} \text{ with } s : T+X \xrightarrow{a} p \text{ with } \$.$$

Hence by 4.1,

$$\vdash \xi, s'.\ \langle gc, s\rangle \xrightarrow{a} \langle \xi, s'\rangle\ R_{T+X}(commands(gc))\ p \text{ with } \$$$

since a straightforward inductive proof shows that almost-terminal configurations <done,$> and <abort,$> cannot result since a≠1; so

$\langle do\ gc\ od, s\rangle \xrightarrow{a} \langle \xi; do\ gc\ od, s'\rangle\ R_X(c)[2.1]\ r$

5.2 Suppose

$$[\![gc]\!]_{T+X} \text{ bef } D_{gc} \text{ with } s : T+X \xrightarrow{1} r$$

then, at first sight, we have 3 cases;

1. $[\![gc]\!]_{T+X}[sub(T+X,s)] \xrightarrow{DONE!} p,\ r = p \text{ bef } D_{gc}$ with s :

   which implies $\exists\ \xi \in commands(gc)$ such that

   $[\![\xi]\!]_{T+X}[sub(T+X,s)] \xrightarrow{DONE!}$

   contradicting the inductive hypothesis (2)

2. $[\![gc]\!]_{T+X}[sub(T+X,s)] \xrightarrow{ABORT!}$: is treated similarly.

3. $[\![gc]\!]_{T+X}$ with $s : T+X \xrightarrow{1} p$ with \$,

   $r = p$ bef D_{gc} with \$,

   then by 4.2 $\exists$ y such that

   $\langle gc, s\rangle \xrightarrow{\varepsilon} y\ R_{T+X}(commands(gc))\ p$ with \$

   We proceed by cases on y;

   1. $y = \langle done, s\rangle$: then

      $null(p')$, $s = \$$

      so $\langle SKIP, s\rangle$ END p with \$

      whence

      $\langle SKIP, s\rangle\ R_{T+X}(commands(gc))\ p$ with \$,

      $\langle do\ gc\ od, s\rangle \xrightarrow{\varepsilon} \langle SKIP; do\ gc\ od, s\rangle,\ R_X(c)[2.1]\ r$

   2. $y = \langle abort, s\rangle$: similarly.

   3. $\exists\ \xi', s'.\ y = \langle \xi', s'\rangle$: then immediately

      $\langle do\ gc\ od, s\rangle \xrightarrow{\varepsilon} \langle \xi'; do\ gc\ od, s\rangle,\ R_X(c)[2.1]\ r$

case $c = c_1 || c_2$:

1. Those required pairs are contained in [1].

2. The proof is very like that for the sequential command and will not be given here.

3. The transitions may be analysed as follows;

For 3.1 :

$[1] \xrightarrow{a} [1]$ (a,b,c) Action other than termination.

$[1] \xrightarrow{1} [2]$ (a) One command is SKIP.

$[1] \xrightarrow{1} [3]$ (a) One command is ABORT.

$[2] \xrightarrow{a} [2]$ (a,b,c) Executing command ξ.

$[2] \xrightarrow{1} END$ (a) ξ = SKIP or ξ = ABORT.

$[3] \xrightarrow{a} [3]$ (a,b,c) Executing command ξ.

$[3] \xrightarrow{1} END$ (a) ξ = SKIP or ξ = ABORT.

For 3.2 :

$[1] \xrightarrow{a} [1]$

$[1] \xrightarrow{\varepsilon} [2]$

$[1] \xrightarrow{\varepsilon} [3]$

$[2] \xrightarrow{a} [2]$

$[2] \xrightarrow{\varepsilon} END$

$[3] \xrightarrow{a} [3]$

$[3] \xrightarrow{\varepsilon} END$

We take as an exemplary case pairs in [1], so suppose

p_1 par p_2 with \$ $\xrightarrow{a}$ r

and let $R_1 = R_X(c_1)$, $R_2 = R_X(c_2)$.

We proceed by cases on the derivation;

1. p_1 communicates with the simulated state, i.e.

$\exists$ α, p_1', \$' such that

$p_1 \xrightarrow{\alpha} p_1'$, $[\![\$]\!] \xrightarrow{\bar{\alpha}} [\![\$']\!]$,

a = 1, r = p_1' par p_2 with \$'

then by sorts $\exists$ $\$_1'$ such that

$[\![\$_1]\!] \xrightarrow{\bar{\alpha}} [\![\$_1']\!]$, $\$_1'*\$_2 = \$'$

so

p_1 with $\$_1 \xrightarrow{1} p_1'$ with $\$_1'$

and hence by the inductive hypothesis either

$$\langle \xi_1, s_1 \rangle \; R_1 \; p_1' \text{ with } \$_1'$$

or

$$\exists\, y.\ \langle \xi_1, s_1 \rangle \xrightarrow{\underline{\varepsilon}} y \; R_1 \; p_1' \text{ with } \$_1'$$

In the former case

$$\langle \xi_1 || \xi_2, s \rangle \; R_X(c)[1] \; r$$

and in the latter case we proceed by cases on y;

1. $y = \langle done, s_1 \rangle$: then

$$nuld(p'),\ \$_1' = s_1,\ \xi_1' = SKIP$$

from which

$$\langle \xi_1, s_1 \rangle \text{ END } p_1' \text{ with } \$_1'$$

so we establish (b) since

$$\langle \xi_1 || \xi_2, s \rangle \; R_X(c)[1] \; r$$

2. $y = \langle abort, s_1 \rangle$: is handled similarly.

3. $\exists\, \xi_1', s_1'.\ y = \langle \xi_1', s_1' \rangle$: then immediately

$$\langle \xi_1' || \xi_2, s' \rangle \; R_X(c)[1] \; r,\ s_1' * s_2 = s'$$

2. p_1 sends a value to p_2, i.e. $\exists$ A, v, p_1', p_2' such that

$$p_1 \xrightarrow{\underline{A!v}} p_1',\ p_2 \xrightarrow{\underline{A?v}} p_2',\ a=1,$$

$$r = p_1' \text{ par } p_2' \text{ with } \$$$

then

$$p_1 \text{ with } \$_1 \xrightarrow{\underline{A!v}} p_1' \text{ with } \$_1,$$

$$p_2 \text{ with } \$_2 \xrightarrow{\underline{A?v}} p_2' \text{ with } \$_2$$

and hence by the inductive hypothesis $\exists\, \xi_1', \xi_2', s_2'$

$$\langle \xi_1, s_1 \rangle \xrightarrow{\underline{A!v}} \langle \xi_1', s_1 \rangle \; R_1 \; p_1' \text{ with } \$_1$$

$$\langle \xi_2, s_2 \rangle \xrightarrow{\underline{A?v}} \langle \xi_2', s_2' \rangle \; R_2 \; p_2' \text{ with } \$_2$$

whence

$$\langle \xi_1 || \xi_2, s \rangle \xrightarrow{\underline{\varepsilon}} \langle \xi_1' || \xi_2', s' \rangle \; R_X(c)[1] \; r,$$

$$s_1 * s_2' = s'$$

3. p_1 acts on its own, i.e. $\exists\, p_1'$ such that

   $p_1 \xrightarrow{a} p_1'$, $r = p_1'$ par p_2 with \$

   then

   p_1 with $\$_1 \xrightarrow{a} p_1'$ with $\$_1$

   and from here on we proceed as in case 1.

4. p_1 terminates successfully, i.e. $\exists\, p_1'$ such that

   $p_1 \xrightarrow{DONE!} p_1'$, $r = p_1'$ parD p_2 with \$, a=1

   then

   $null(p_1')$, $s_1 = \$_1$, ξ_1 = SKIP

   whence we establish (a) since

   $\langle \xi_1 || \xi_2, s\rangle \xrightarrow{\varepsilon} \langle \xi_2; SKIP, s\rangle\ R_X(c)[2]\ r$

   and

   $\$ = \$_1 * \$_2 = s_1 * \$_2 = s * \$_2$ (since $s_1 * s_2 = s$)

5. p_1 terminates unsuccessfully; rather like case 4.

The (symmetric) cases for p_2 are addressed in exactly the same fashion.

4. Since c ($=c_1||c_2$) is neither an alternative nor an iterative command the result is immediate.

5. Let $T = TV(c_1) + TV(c_2)$;

   5.1 Suppose

   $[\![c_1]\!]_{T+X}$ par $[\![c_2]\!]_{T+X}$ with s :$T+X \xrightarrow{a} r$, $a \neq 1$

   then since $a \neq 1$ we must have $\exists\, p_1, p_2$ such that

   $r = p_1$ par p_2 with \$

   $[\![c_i]\!]_{T+X}$ with s :$T+X \xrightarrow{a} p_i$ with \$,

   $p_j = [\![c_j]\!]_{T+X}$ [sub(T+X,s)],

   $\{i,j\} = \{1,2\}$

Now we apply the inductive hypothesis to get

$\langle c_i, s\rangle \xrightarrow{a} y\ R_{T+X}(c_i)\ p_i$ with \$

The proof proceeds routinely by cases on y.

5.2 Suppose

$$[\![c_1]\!]_{T+X} \text{ par } [\![c_2]\!]_{T+X} \text{ with } s : T+X \xrightarrow{1} r$$

then by the inductive hypothesis (2) neither translated command can have signalled termination, so

$$r = p_1 \text{ par } p_2 \text{ with } \$$$

Now we have two cases,

1. $[\![c_i]\!]_{T+X}$ with $s : T+X \xrightarrow{a} p_i$ with \$,

   $[\![c_j]\!]_{T+X}$ with $s : T+X \xrightarrow{\bar{a}} p_j$ with s, $a \neq 1$

   $\{i,j\} = \{1,2\}$:

   Then by the inductive hypothesis $\exists\, \xi_i, \xi_j$.

   $$\langle c_i, s\rangle \xrightarrow{a} \langle \xi_i, s'\rangle \; R_{T+X}(c_i)\; p_i \text{ with } \$$$

   $$\langle c_j, s\rangle \xrightarrow{\bar{a}} \langle \xi_j, s\rangle \; R_{T+X}(c_j)\; p_j \text{ with } s$$

   so

   $$\langle c_1 || c_2, s\rangle \xrightarrow{\varepsilon} \langle \xi_1 || \xi_2, s'\rangle \; R_X(c)[1]\; r$$

   and we can use [1].

2. $[\![c_i]\!]_{T+X}$ with $s : T+X \xrightarrow{1} p_i$ with \$,

   $p_j = [\![c_j]\!]_{T+X}\; [\mathrm{sub}(T+X, s)]$:

   Similarly.

□

This concludes the (fragment of) the proof for Theorem 3.2. It includes the main concepts and techniques required to deal with the remaining cases.

Corollary 3.4

$[\![.]\!]$: config(CSP) -> config(L[CCS]) is result-good.

□

4. A testing theory of translation correctness

A basic problem of translation theory is to determine the weakest properties required of a translation in order that there is never any conclusive disagreement over the properties of an object and its translation. To formalise this notion requires a more careful examination of the notions of 'conclusive disagreement' and 'properties'.

As with the bisimulation theory we shall attempt to generalise an existing equivalence relation over CCS; this time we employ the 'testing' equivalence of [de Nicola and Hennessy 84].

4.1. Motivations

Given any system it is a minimal requirement that we have some means of testing, i.e. observing, its properties. Typically this observation may extend over a number of interactions and constitute an interactive dialogue with the system, or it may be a simple static test, or it may be a combination of the two. However, it seems neither necessary nor desirable to demand that the tests performed on the target be identical with those performed on the source; in fact we just need a 'corresponding' test. For example you may simulate the action of an electronic digital computer on a piece of paper; tests on the source will involve button pushing, reading VDU's etc. while those on the target involve rifling through bits of paper. Correctness of a translation will depend on a preservation of certain relationships between objects and the tests performed upon them.

In this section we aim to determine very weak requirements for translation correctness by analysing what is required of a translation in order that we are assured of complete and lasting agreement over the properties of the source objects and their translations. Essentially there are two key points on which the theory will rest; firstly we will not require the existence of a single 'super-observer' to compare the behaviour of source and target objects. Such an approach, which is essentially that taken by the bisimulation theory,would, for example, be difficult to apply in cases where a single observable step of the source object is translated to an action involving many observable steps of the target. Instead it is much simpler to postulate the existence of a 'degenerate' observer monitoring the outcomes (only) of experiments , where an experiment is the application of a relevant test to some object. Relevant tests of course will vary according to the object. For example if the source objects are variously coloured and flavoured gob-stoppers while the target objects are a set of similarly coloured billiard balls then tests for flavour are relevant in the source but not the target. (However, it may be that tests for flavour can be 'coded up' as tests for colour, eg. 'is-it-strawberry' becomes 'is-it-red'). Thus relevance is determined by applicability and the ability to discriminate between objects presented for examination.

All we will require of observers is that they can distinguish whether the result of some experiment was a success or a failure. We will not distinguish modes of failure in experiments, such as the usual notions of deadlock and divergence. If an experiment does not succeed it just does not succeed; we do not say that it

is deadlocked or divergent or anything else. Note this applies to experiments but not necessarily the objects undergoing testing, for given acute enough tests we may be able to distinguish deadlock and divergence within process objects.

4.1.1. Experiment systems

The theory of experimentation outlined above intentionally does not determine the allowable tests, notions of success, or how tests are to be administered. Thus any particular set of experiments will be parameterised on particular instantiations for these variables.

Definition 4.1

An <u>experiment system</u> Ex is a triple <E,->,success> where

E is a set of 'experiments', ranged over by e.

-> $\underline{C}$ E x E, is a binary relation (i.e. an unlabelled transition relation).

success $\underline{C}$ E, is a set of 'successful' experiments.

In defining or analysing an experiment system Ex we may write

exp(Ex) for E

trans(Ex) for ->

succ(Ex) for success.

□

Thus an experiment system is very similar to a labelled transition system with just one label used all the time, no set of terminals, but with an added notion of success. Given the similarity between

<E, →,success> and a labelled transition system <E,->,{ε},φ> we will import many of the notions (such as computations, derivatives etc.) from labelled transition systems without further comment.

Though for much of the theory to be developed it is not necessary to distinguish test from object in an experiment, it is intuitively expected that some test is being performed on some object in order to answer some question. Thus in many cases E will be described in the form

E = T[P]

where

T is a set of <u>tests</u>, ranged over by t.

P is a set of <u>objects</u>, ranged over by p.

T[P] = {t[p]: applicable(t,p)}.

t[p] is the experiment formed by interfacing test t with object p in some way to be defined.

applicable is a binary relation governing whether or not some test t and object p can be interfaced. If all tests are applicable to all objects then its definition may be omitted.

As an example construction of an experiment system we show how, in the spirit (if not the letter) of [de Nicola and Hennessy 84], CCS may be adapted to this paradigm.

Example EXP(CCS)

We proceed by giving the definitions of experiments, transitions between experiments and success, thereafter turning to their motivation. An experiment is a CCS term, that is we take

exp(EXP(CCS)) = {t|p: t,p ε Terms, ⊢t, ⊢p}

assuming as in chapter 1 a set of action names Act, a set of procedure names Proc, and a definition set D. We may write t[p] for the experiment t|p.

Recalling the definition of the transition relation -> $\subseteq$ Terms x (|1|+labels(L[CCS])) x Terms from chapter 1 we introduce the successful experiments by assuming a distinguished action $\omega \in$ Act and defining

succ(EXP(CCS)) = {e: e $-\underline{\omega!}\rightarrow$ }

An experiment proceeds by the (externally) silent transitions of the CCS term,

trans(EXP(CCS)) = $-\underline{1}\rightarrow$

Given that we have made a distinction between test and object in an experiment t[p] what is being tested for? The idea is that t is designed to test (or use) some aspect of p's behaviour, and the experiment is constructed to answer the question "will p satisfy t?". For example take

t = A!.B!.ω! , p = A?.B? + A?.C?

The question then becomes "will p do a receive action on channel A before a receive action on channel B", and the answer is "maybe". How answers are derived and used to compare objects is the subject of the remainder of this chapter.

Note, if the test was 'wrongly' applied, say as

t[p] = t + p

then you have an answer to a different question. The art of constructing interesting experiments lies in asking the right questions; for CCS the most interesting questions address the communication behaviour of a process so experiments of the form

t|p (in the above system) will provide the most information, t having free access to p. In a sequential language like Pascal processes don't communicate except by altering the state for a subsequent process; thus you might expect experiments of the from p;t where t tests the state resulting from the execution of p. Thus, in summary, experiments frame questions about the properties of their constituent objects. Questions thus posed have answers in the set true, false and maybe.

Returning to the question of where we draw the line between test and object we can see that in the design of most experiments some distinction is intended. However, the progress of an experiment may make no use of that distinction.

For example an experiment to measure the temperature of a phial of ice water by immersing a thermometer must consider the possibility that the object whose temperature is measured is not the body of water alone but rather the water plus a warm thermometer. In CCS t|(p|q) behaves just like (t|p)|q, so is the test t or t|p? We will find this fuzzy distinction between test and object (in EXP(CCS)) very useful in chapter 6.

We end this example by noting that part of the art of designing experiments lies in ensuring that everything determining the outcome is described in the experiment; an experiment should represent a "closed world". This is reflected in our definition of trans(EXP(CCS)) and is crucial to the definition of implementation to follow.

□

4.1.2. Translations

The position we adopt now is that a translation specifies a mapping between experiments rather than the more usual mapping between objects to be experimented upon, so a translation now contains information not only on the correspondence between objects but also on that between tests on those objects.

Definition 4.2

Let EX1 = $\langle E, \rightarrow_1, success_1 \rangle$, EX2 = $\langle F, \rightarrow_2, success_2 \rangle$ be experiment systems. A translation is a total function $[\![.]\!]$: E -> F.

□

Note that in this general definition a translation is not required to give any information on what objects are being experimented upon or a correspondence between them.

For the remainder of this chapter we assume the pair EX1 and EX2 of experiment systems, e ranging over E and f over F, together with a translation $[\![.]\!]$: E -> F. It is expected that most translations between experiment systems will be generated in a context-free form, that is if E = $T_1[P_1]$ and F = $T_2[P_2]$, then

$$[\![\, t[p] \,]\!] = [\![t]\!]_T[[\![p]\!]_P]$$

where $[\![.]\!]_T : T_1 \rightarrow T_2$,

$[\![.]\!]_P : P_1 \rightarrow P_2$

are total functions and applicable1(t,p) implies applicable2($[\![t]\!]_T$, $[\![p]\!]_P$)

However, this is <u>not</u> necessary and indeed the translation of chapter 5 is not of this form.

4.2. Translation correctness

Our intention is that a correct translation should preserve certain properties of experiments or, equivalently in many cases, certain relationships between objects and tests. However, we must appreciate there is some measure of asymmetry between the two experiment systems when judging a translation. This is because the target object is intended as a "faithful transcription" of the source object, but the converse is not in general necessary. Thus we assume the observer has expectations of the properties of the target object discernable by testing; how may he be disappointed?

4.2.1. Games observers play

Translation correctness can be viewed as a two player game in the everyday informal sense. The two players are called the implementer and the observer and the aim of the game (for the observer) is to find mistakes in a translation supplied by the implementer. The aim of the game for the implementer is to avoid losing to the observer.

The field of play is composed of the following entities, provided by the implementer:

1. A set of source experiments, ranged over by e.

2. A set of target experiments, ranged over by f.

3. A translation $[\![.]\!]$ from source experiments to target experiments so that $[\![e]\!]$ is claimed to 'implement' e

satisfactorily (i.e. without mistakes).

4. A specification which associates with each source experiment e a set of "computation descriptions", one for each possible computation of e (there being at least one such). A computation description may take many forms, for example "the red light will flash and then it stops", or "successful" etc.

5. A relation corresponds between descriptions of source experiment computations and descriptions of target experiment computations, e.g.

"the red light will flash and then it stops"

corresponds

"the green light goes out and then it explodes".

The implementer having provided all these things the game proceeds as follows:

A. The observer picks a source experiment e.

B. The observer then performs the target experiment [[e]] to get a target computation (description) and tries to produce a "corresponding" source computation, using **corresponds** to get a source computation (description) and looking for that in the specification of the possible computations of e.

If no such computation can be found the observer wins.

C. Goto A.

So the game played is essentially

> Given a computation of the target experiment produce a "corresponding" source experiment computation from the specification.

Thus the observer must be able to justify the observed computation as corresponding in some way to a particular source computation, saying "This computation I have observed is <u>really</u> that computation" while pointing out the correspondent in the source specification.

Now, crucial to the notion of implementation is the assumption (stated here for the first time) that the observer has no control over the particular computation exhibited during the performance of a target experiment. That is, if an experiment has more than one possible computation then any factors which may determine a particular computation (e.g. the phase of the moon, ambient temperature etc.) are assumed to be unknown to the observer, so to the observer the experiment may appear to be non-deterministic.

Hence we need not require

> Given a computation from the specification of the source experiment produce a "corresponding" target experiment computation.

because although we have a specification for the source to which the observer points while justifying the target computation there is no specification for the target which can be pointed to. The only way to produce a target computation to point at is to 'force' the target experiment to produce it. But the observer cannot force it, whenever an experiment runs whatever choices it

has cannot be influenced by the observer.

Perhaps a simple, all too familiar, example will clarify the point:

> Your TV set is not working properly; sometimes when you turn it on it works fine, but sometimes it just sits there and fizzles. You seem to have no way of guessing which it will do when you go to turn it on and you cannot divine any way of influencing the outcome. You phone the TV repairman who dutifully rushes to your aid.
>
> "What's the problem?" he says, asking for a specification of what the TV set does.
>
> "Sometimes when I switch it on it works just fine, sometimes it just fizzles" you say, giving him the specification in terms of the experiment to be performed and the, two, possible computations thereof.
>
> "OK mate, lets give it a go" he says, adopting the role of observer now he has the experiment and the specification. He will now attempt to verify that the specification and the implementation (your TV set) correspond.
>
> He switches it on, it works fine. Try again: he switches it off and then on again. Still works fine. Again. Again.... He performs the same experiment many times without observing the computation which fizzles; he is unable to force its occurrence.

Is he justified in denouncing you as a liar? Clearly not, you said it might work and it just so happens that it did so every time. His inability to produce the fizzle is no evidence against the validity of the specification, everything that happened was specified as a possibility. (Incidentally, you might try employing him to switch your TV set on every night; he seems, inadvertantly, able to determine <u>your</u> desired computation).

The field of play for the TV set game is:

1. There is only one source experiment, namely the hypothetical switching on of your TV set in a conceptual world.

2. For the single source experiment the description of its allowable computations is:

   "Sometimes it works just fine, sometimes it just fizzles"

3. The single target experiment is the actual switching on of the actual TV set in the real world.

Note we will make the normal simplifying assumption that switching the TV set on and then off has no lasting effect on the TV set, so that each time it is switched on again it is a new experiment rather than a continuation of the old one. This assumption seems to be made by the repairman also.

4. With only one source experiment and one target the translation is clear.

5. The relationship <u>corresponds</u> is that between the description of a phenomenon and the phenomenon itself.

Returning to the general game of translation correctness we now spare a thought for the unhappy lot of the implementer; he cannot win. However, we will define translation correctness in such a way as to guarantee he doesn't lose. Firstly we need to fix ourselves more firmly in the domain of experiment systems by defining the notion of 'correspondence' to be used for computations, and we begin that by extending the notion of success from experiments to experiment computations.

Definition 4.3

Suppose EX = <E,->,success> is an experiment system and $\langle e_k \rangle_K \in$ comp(EX) is a computation, finite or infinite, in this system. Then define

$$\text{success}(\langle e_k \rangle_K) \text{ iff } \exists\, k \in K.\ \text{success}(e_k)$$

□.

This seems to be the 'least' definition of success for computations in that we would expect it to imply any other notion.

A specification was said to describe, for each experiment e, its allowable computations in some way. The way we have chosen to describe experiment computations is by their success or failure; so a specification is a pairing of experiments with their possible results.

Definition 4.4

Let EX = <E,->,success> be an experiment system, $e \in E$, $c \in$ comp(e). Then define

$$\text{result}(c) = \begin{cases} \{\top\} & \text{if success}(c) \\ \{\bot\} & \text{otherwise} \end{cases}$$

and define the result-set Res(e) by

$$Res(e) = \sum_{c \in comp[e]} result(c)$$

Thus a result-set is the set of all possible outcomes (success or failure, $\top$ or $\bot$) of an experiment. Finally, define

$$Ressets = \{\{\top\}, \{\bot\}, \{\top, \bot\}\}$$

the set of all possible result-sets, ranged over by R.

□

It may appear at first sight that we have chosen to largely ignore the role computation descriptions might play in the general theory by restricting them to one bit (success or failure) of information. However, for our purposes (and probably most others) it will suffice when the tests employed are sufficiently rich to constitute descriptions of object computations. Then an experiment succeeds when the object performs a computation from a set described by the test.

Proposition 4.1

Let $EX = \langle E, \rightarrow, success \rangle$, then $\forall e \in E.\ Res(e) \in Ressets$

Proof It is merely necessary to prove $R \neq \phi$, which follows immediately since the set comp(e) is always non-empty.

□.

The form of specifications is now clear; they define for each experiment e its result-set, a description of its possible computations.

Definition 4.5

Let $EX = \langle E, \rightarrow, success \rangle$, then a specification $spec \in SPEC(EX)$ is a pairing of experiments with their possible results:

SPEC(EX) = E -> Ressets

□

Note that since specifications always describe hypothetical experiments we can take the specification to <u>be</u> the result-set function Res for these experiments.

Having fixed the descriptions of computations in a vocabulary of success and failure common to all experiment systems the notion of correspondence becomes trivial, we can take it to be equality over $|T, \bot|$. Thus T <u>corresponds</u> T, $\bot$ <u>corresponds</u> $\bot$ and no other correspondences hold. In what follows explicit reference to the correspondence relation will usually be dropped since it has been reduced to equality.

Returning to the plight of the implementer we recall that he loses when the observer produces a target computation description which has no correspondent in the specification, that is

$$\exists\, e \in E,\ c \in comp([\![e]\!]).\ result(c) \notin Res(e)$$

To avoid this possibility we introduce the notion of 'implementation' in the following definition;

Definition 4.6

f implements e iff $Res(f) \subseteq Res(e)$

□

This is the major definition of this chapter.

4.2.2. Comparing results

It will be helpful, both technically and intuitively, to restate the definition of 'implements' in terms of two predicates, must and may, over experiments corresponding to the necessity and possibility (respectively) of a successful outcome to the experiment. We also employ a predicate mustnot to express necessity of failure, but it will become redundant later. All predicates are written postfix.

Definition 4.7

Let e ε E, then

e must iff Res(e) = {⊤}

e mustnot iff Res(e) = {⊥}

□

We can express implementation in terms of must and mustnot as the following proposition shows,

Proposition 4.2

f implements e

iff

1. e must implies f must

2. e mustnot implies f mustnot

Proof Suppose f implements e, then

Res(f) ⊆ Res(e)

1. By the definition of must

e must implies Res(e) = {⊤}

and, by Proposition 4.1

Res(f) ε Ressets

so Res(f) = {⊤}.

2. Similarly.

Now suppose 1 and 2; clearly if e must or e mustnot then Res(e) = Res(f). Otherwise the result is trivially necessary since then Res(e) = {⊤,⊥}.

□

It is easy to see that at least the forward impication is well-founded in intuition; returning to our earlier TV set example if you say the set must fizzle and it doesn't then the set is not how you described it. Similarly if you say it mustnot work then it better hadn't. (This should bring us to the conclusion that when constructing specifications implementers would like to avoid claims about certainty as much as possible).

To help with the intuitions of the second implication we introduce the predicate may which asserts the possibility of a successful computation,

Definition 4.8

e may iff ⊤ ε Res(e)

□

Proposition 4.3

1. ∀ e. e must implies e may

2. ∀ e. e may iff e must/not

□

Now we can restate Proposition 4.2 as

Proposition 4.4

f <u>implements</u> e iff 1. e <u>must</u> implies f <u>must</u>

2. f <u>may</u> implies e <u>may</u>

□

Thus, essentially, implementation promises two things

1. If success is guaranteed by the specification then it is guaranteed by the implementation.

2. If the implementation succeeds then the specification can too.

4.2.3. Complete implementation

Returning once more to our TV set example we notice that a fully functional TV set which always works implements the specification given to the repairman (as does one which always fizzles). Thus we see that the repairman believes the set is more than just an implementation in our defined sense, he believes that every result possible in the specification is possible in the implementation. Hence we would like to reassure the repairman by telling him not just that it implements the specification, but rather that it <u>completely</u> implements the specification.

Definition 4.9

f <u>completely-implements</u> e iff Res(f) = Res(e)

□

Thus the TV set which always works is not a complete-implementation of the specification "sometimes it works, sometimes it just fizzles" given to the repairman.

We can recast this definition in a number of ways,

Proposition 4.5

The following statements are equivalent:

1. f <u>completely-implements</u> e
2. f <u>implements</u> e, e <u>implements</u> f
3. e <u>must</u> iff f <u>must</u>, e <u>may</u> iff f <u>may</u>

□

Two sorts of contract are now possible between implementer and observer, viz. "it is an implementation" and "it is a complete-implementation", but it should be noted here that the observer will never (within the rules and conditions of the game as given) be able to show an implementation is not complete. An extended version of the game might involve playing the game the other way around by taking computations of the specification and demanding the implementation produce them; inability to do so would show the implementation was not complete.

4.2.4. Translation correctness

With the two available forms of contract between observer and implementer come two notions of translation correctness, one of which subsumes the other.

Definition 4.10

Let $[\![.]\!]$: E -> F be a translation, then

1. $[\![.]\!]$ is an <u>implementation</u> iff $\forall$ e $\in$ E. $[\![e]\!]$ <u>implements</u> e
2. $[\![.]\!]$ is a <u>complete-implementation</u> iff

   $\forall$ e $\in$ E. $[\![e]\!]$ <u>completely-implements</u> e

□

Proposition 4.6

Any complete-implementation is an implementation.

□

Probably one of the most important properties we would like for any notion of translation correctness is compositionality, that is if we translate A to B and B to C both correctly then the translation A to C is correct.

Proposition 4.7

Suppose $[\![.]\!]_1$:E -> F and $[\![.]\!]_2$:F -> G are translations,then

1. $[\![.]\!]_1$, $[\![.]\!]_2$ both implementations implies

   $[\![.]\!]_2 \circ [\![.]\!]_1$: E -> G is an implementation.

2. $[\![.]\!]_1$, $[\![.]\!]_2$ both complete-implementations implies

   $[\![.]\!]_2 \circ [\![.]\!]_1$: E -> G is a complete-implementation.

□

4.2.5. Implementations in CCS

The notion of implementation has, so far, been concerned solely with experiments but we now extend it so that we may say one object implements another rather than one experiment implements another. Clearly we are now going to distinguish between tests and objects in experiments so suppose

$$Ex = \langle T[P], \rightarrow, success \rangle$$

and assume (for convenience) all tests $t \in T$ are applicable to all objects $p \in P$.

An object p implements another object q if every application of a test to p implements the experiment applying the same test to q.

Definition 4.11

1. p <u>implements</u> q iff ∀ t ε T. t[p] <u>implements</u> t[q].

2. p <u>completely-implements</u> q iff
   ∀ t ε T. t[p] <u>completely-implements</u> t[q].

□

Proposition 4.8

1. <u>implements</u> is a preorder over P.

2. <u>completely-implements</u> is an equivalence over P.

□

Examples

We can now present a few example implementations of processes together with an experiment system EXP1 similar to EXP(CCS) but restricted in a way appropriate to the examples. The examples concern the implementation of a simple sorting machine defined in CCS by

$$sort(M) \Leftarrow user?x.sort(M{+}{+}[x]) + \text{if } M = \phi \text{ then } user!min(M).sort(M{-}{-}[min(M)])$$

where M ranges over multisets of natural numbers, and min(M) is the smallest number occurring in M.

Thus a sorting machine is always willing to receive from a user another value to add to its collection , and if its collection is non-empty then it is prepared to deliver to the user the smallest value therein, revising its collection appropriately in both cases.

The experiment system EXP1 is designed to allow only very restricted testing of 'sort(M)' and its implementations, for reasons discussed in example 1 which follows the definition of EXP1.

Definition EXP1

Suppose L[CCS] is the transition system for CCS containing all the action names, procedure names and definitions used in the following examples.

The objects to be tested are CCS process descriptions without the distinguished action name ω:

$$P = \{p:\ p \in config(L[CCS]),\ \omega \notin FL(p)\}$$

A test follows a 3 step cycle:

1. As a user offer a value to the machine.
2. As a user accept a value from the machine.
3. Decide,on the basis of some boolean test, whether or not to continue.

We allow a test to succeed by offering ω, so

$T \subseteq config(L[CCS])$, ranged over by t and defined by

t ::= user!n.user?x.if b then t | 1.ω!

where n ranges over numerals for natural numbers.

A test is interfaced by putting it in parallel with the object and both test and object must be well-defined terms:

$t[p] = t|p \in config(L[CCS])$ if $\vdash t$ and $\vdash p$

An experiment proceeds by the usual CCS semantics:

$t[p] \rightarrow t'[p']$ iff $t|p \xrightarrow{1} t'|p'$

Success, as in EXP(CCS) is an offer of ω

success(t[p]) iff $t[p] \xrightarrow{\omega}$

Now, EXP1 = <T[P],->,success>.

□

Example 1

Whereas sort(ϕ) is a completely general sorter for multisets of arbitrary size we may only be able to implement a machine of some fixed finite size. For example we can define a degenerate sorting machine for multisets of no more than one element; basically just an "echo machine":

sort1 <= user?x.user!x.sort1

so sort1 can only receive a value and then immediately output it.

Now, since <u>implements</u> is defined relative to a set of tests (or more precisely relative to an experiment system,see Definition 4.11) we see that in EXP(CCS) sort1 does not implement sort(ϕ). For example (in EXP(CCS)) take the test

t = user!1.user!2.1.ω!

then t[sort(ϕ)] <u>must</u>

but t[sort1] <u>mustnot</u>

However, restricting the applicable tests as in EXP1 corresponds to "conditions of use" under which the object <u>will</u> function as an implementation, and in EXP1 sort1 <u>implements</u> sort(ϕ) since the "conditions of use", enshrined in the definition of tests, state that the machine will never be asked to hold more than one value at a time.

In general then, <u>implements</u> is defined relative to an experiment system EX in which the conditions of use are implicit in the

structure and composition of the experiments allowed. For the statement 'p implements q' to be of use it must be shown that p will be used in an experiment of EX.

Example 2

In this example we will see that if a specification allows a choice to be made in an 'uncontrollable' way then an implementer is perfectly within his rights to make an arbitrary choice once and for all. Consider the, rather odd, sorting machine

$$\begin{aligned} sort2(M) \Leftarrow\ & user?x.sort2(M{+}{+}[x]) \\ & + \\ & user?x.user!m.sort2(M{+}{+}[x]{-}{-}[m]) \\ & + \\ & \text{if } M = \phi \text{ then } user!min(M).sort2(M{-}{-}[min(M)]) \end{aligned}$$

where $m = min(M{+}{+}[x])$. This is very similar to 'sort' except that when receiving a new value it can arbitrarily decide to make the users next interaction a receipt of the smallest value ($m = min(M{+}{+}[x])$) currently held. The somewhat churlish behaviour of $sort2(\phi)$ is implemented (but not completely implemented) by sort1 in both EXP1 and EXP(CCS) since sort1 always makes its (potentially arbitrary) choice in favour of the second summand above. This is closely analogous to the TV set example given earlier in that a TV set that always works when switched on implements one that also has the (uncontrollable) option of fizzling. This sort of implementation will be addressed more fully in chapter 6.

Example 3

It is possible, in the appropriate conditions, for certain forms of divergence to prevent one process implementing another. The

appropriate conditions will depend on the form of the divergence and would typically constitute conditions on the form of experiment systems. Rather than trying to pursue a general approach we will merely give an example of two processes which are bisimulationally equivalent but only one of which implements the other in EXP(CCS).

$$sort3(M) \Leftarrow user?x.sort3(M{+}{+}[x]) + 1.sort3(M) + \text{if } M \neq \phi \text{ then } user?min(M).sort3(M--[min(M)])$$

then, recalling $\approx$ from chapter 2,

$$sort(\phi) \approx sort3(\phi)$$

but $sort3(\phi)$ does not implement $sort(\phi)$ in either EXP1 or EXP(CCS) since for example in EXP1,

$$1.\omega! \mid sort(\phi) \ \underline{must}$$

but

$$1.\omega! \mid sort3(\phi) \ \underline{mu/st}$$

since

$$1.\omega! \mid sort3(\phi) \rightarrow 1.\omega! \mid sort3(\phi) \rightarrow \ldots$$

forms an infinite unsuccessful computation.

4.3. Summary and comparison

In this chapter we have motivated and described an approach to translation correctness based on the testing equivalence for CCS of [de Nicola and Hennessy 84]. Translations were defined as functions between experiments. Thus rather than just translating objects we also require the "conditions of use" to be translated also. A careful analysis of the aims of correctness criteria

resulted in two forms of correctness being defined: implementation and complete-implementation.

It now seems appropriate to compare the theories of translation correctness so far described. Each theory takes a very different approach to defining translation correctness, making a technical comparison both very difficult and of doubtful value. Rather we seek here to illuminate various points where the testing theory differs substantially in approach. We begin with the bisimulation theory:

4.3.1. A comparison with the bisimulation theory

We divide the comparison into two parts: what you gain and what you lose.

Gains by using the testing theory

1. Freedom from fixed actions: the bisimulation theory presupposes a common set, or at best two sets with a bijective mapping between them, of action labels. In the definition of bisimulations (chapter 2 section 1) we find a rigid demanding form of comparison which is far too strict for many reasonable translations; an example of such a translation (and an analysis of why the bisimulation approach won't work) is the subject of chapter 5 where the aim of the translation is the eradication of action labels.

2. Freedom from fixed choice structure: the bisimulation mode of comparison essentially demands that any 'processes' paired in a bisimulation have exactly the same capabilities. As was

shown in [de Nicola and Hennesssy 84] this can lead to processes with intuitively the same behaviour having no bisimulation. For example, suppose

$$p = A!.B! + A!.C!, \quad q = A!.(1.B! + 1.C!), \quad p\ R\ q$$

for some $R \subseteq Terms \times Terms$ a bisimulation, then we can derive a contradiction since

$$A!.(1.B! + 1.C!) \xrightarrow{A!} 1.B! + 1.C!$$

but

$$A!.B! + A!.C! \xrightarrow{A!} r \text{ implies } r = B! \text{ or } r = C!$$

and clearly in either case $(r, 1.B! + 1.C!)$ cannot be a member of a bisimulation since r will lack some capability of $1.B! + 1.C!$ ($r \xrightarrow{B!}$ or $r \xrightarrow{C!}$ but not both). The point is that p has made an (uncontrollable) choice sooner than q, the choice being which of B!, C! to offer after A!. This 'motion' of choice points will occur in the translation of chapter 5 and will be carefully analysed therein.

3. Freedom to implement: the relation <u>implements</u> over process objects (Definition 4.11) is a pre-order rather than an equivalence like = over CCS terms. This can make life easier for the implementer in two ways; firstly given a specification he need not <u>fully</u> implement it, as was seen in example 2. Secondly, as we will see in chapter 6, given a particular machine he can construct a specification which abstracts away some particular uninteresting property of the machines behaviour while still retaining the relationship of <u>implements</u> between machine and specification.

Losses by using the testing theory

The most significant loss is the bisimulation proof technique, for each of the three gains above are nails in the coffin of any attempt to construct a bisimulation. Gains 2 and 3 cannot possibly allow the construction of a bisimulation but gain 1 can be retained in very special circumstances, of doubtful value, as shown in chapter 5 section 2.1 (where the attempt fails by gain 2 at least). The approach in chapter 6 works only where a direct substitution of specification for implementation in an experiment is possible (so it's not much use for proving translations). A more general approach is pursued in chapter 5 section 2 but it doesn't constitute a proof technique, rather a set of proof 'guidelines' or 'targets' are defined as conditions sufficient to guarantee (complete-) implementation under certain circumstances.

4.3.2. A comparison with Li's theory

Li's approach is very different from that of the testing theory, the notion of observation is crucial to the testing theory but virtually irrelevant to a 'correct' translation in Li's sense. All that matters is that any (and all) tips of the translated configurations computation set correspond to tips in the original configurations computation set (see chapter 2) and each has an infinite computation if the other has. In chapter 5 we will see how the testing theory can take on a very similar appearance, inspecting only the tips of computations for success. However, there are crucial differences, most notably that the testing theory is inspecting computation trees of <u>experiments</u> rather than just object configurations; the observation and influence of the *test part has no counterpart in Li's theory.*

Observation is introduced in the adequacy conditions [Li chapter 5 section 5.3 Definition 5.10] by comparison of action labels, but the example of chapter 5 is not even correct in Li's sense so it cannot satisfy these conditions. A more detailed explanation is given in chapter 5 section 2.

5. A example in the testing theory

The aim of this chapter is to demonstrate a translation which is not 'correct' in any of the operational theories of translation except the testing theory. This is achieved by changing the way communications are performed from the simultaneous matching of complementary labels in the source language to the extended manipulation of a 'communication state' in the target language.

In section 5.1 we first define the source language, going on to define the target language in the light of the desired translation. This defines two labelled transition systems, Source and Target. We then formally present a translation $[\![.]\!]_1$ from configurations of Source to configurations of Target. This translation is of the form employed in all but the testing theory where translations are functions between experiments . In order to apply the testing theory we define two experiment systems, SEXP and TEXP, derived from Source and Target, and extend the translation to these experiment systems as $[\![.]\!]_2$.

In section 5.2 we address the correctness problem, first by establishing that $[\![.]\!]_1$ is not correct for any of the theories given (the testing theory is not directly applicable to $[\![.]\!]_1$). Further, we show, with particular reference to the bisimulation theory, why these theories could not be simply extended to cover $[\![.]\!]_1$. It then remains to articulate the testing theory, proving $[\![.]\!]_2$ is a complete-implementation.

5.1.1. The Source language

The language chosen as the source for the translation is a version of CSP wherein only the very core is retained so we are not concerned with many details largely irrelevant to the aims of this chapter. In particular the following features have been omitted:

1. Nested parallelism - no processes within a parallel construct may contain a parallel construct. This is reflected in the absence of the BNF rule

   $c ::= c_1 || c_2$

   which was present in the CSP of chapter 3. This also removes the need for the 'scoping' construct and, to a lesser extent, the 'renaming' construct.

2. Guarded commands - in anything other than a 'toy' language, which is what we are constructing, some form of conditional construct is necessary. However, it is not necessary to the construction of a translation with the properties we seek and is thus omitted, together with the constructs 'if gc fi' and 'do gc od' employing it.

It should be noted that the removal of the 'do gc od' construct from CSP will have the rather drastic effect of restricting all processes to having only finite computations. This is a serious omission in any language and is tolerated only to simplify presentation and allow the introduction of a set of conditions which, under the finite computations assumption and an assumption of the finality of successful configurations, are sufficient to prove a translation is a complete-implementation.

At the end of this chapter we will see how this technique may be extended to the infinite case.

3. Disjoint naming of processes - in the static semantics for the CSP of chapter 3 the following rule applied a condition 'disjoint(c_1,c_2)' that no two processes in the same parallel construct may have the same process label:

$$11.\quad \frac{\vdash c_1,\ \vdash c_2,\ \text{separated}(c_1,c_2),\ \text{disjoint}(c_1,c_2)}{\vdash c_1 || c_2}$$

We will drop the condition 'disjoint(c_1,c_2)' but retain the others. The reasoning is as follows; given the changes in 1. and 2. the language is now rather too deterministic for our purposes. Removing the disjointness condition introduces nondeterminism by making it possible for there to be many processes competing to make a single communication. For instance the CSP process

<R::S!e || S::R?x || S::R?y , s>

is not syntactically valid for the CSP of chapter 3, but it is upon consideration of such processes that our dismissal of the bisimulation approach will, at least partly, rest.

Note this change is different from 1. and 2. in that it introduces processes that could not be defined in the CSP of chapter 3.

In all other (remaining!) aspects the source language is almost identical with the earlier CSP, any differences will be noted and justified as they occur. We proceed as usual in defining a language, starting with the syntax.

The Source language syntax

As usual we assume the provision of a set Var of variables, a set Proc of process names (ranged over by R,S in this case) and other entities such as states and evaluation functions all as defined in chapter 1 section 1.1.

There are three major syntactic categories:

Acom - a set of 'atomic commands', ranged over by **ac** and defined by

ac ::= x:=e | S!e | R?x

Com - a set of 'commands', ranged over by **c** and defined by

c ::= ac;c | done

Prog - a set of 'programs', ranged over by **p** and defined by

$p ::= R::c \mid p_1 || p_2$

In what follows we will make an heroic attempt to have our cake and eat it with respect to this definition of Prog. Some descriptions of elements p of Prog will follow the above definition but others will proceed as if Prog was defined by

$$p ::= R_1::c_1 \;||\ldots||\; R_n::c_n$$

This flattened form is justified on the grounds that it facilitates many constructions and arguments wherein || may be treated as an, in some appropriate sense, associative operator.

Thus a program describes a set (to most semantic notions || will appear to be commutative as well as associative) of named commands executing, and perhaps communicating via 'instantaneous handshakes', in parallel. Each command is a linear (perhaps empty) sequence of atomic commands ending in 'done', an analogue of Nil in CCS. As with Nil in CCS we will, typically, drop

trailing occurrences of 'done' from commands; thus

x:=e; R?y; done

may be written as

x:=e; R?y

Each atomic command is either an assignment statement (x:=e) or an action to communicate the value of some expression e to any command labelled S (S!e), or else an action to accept the transmission of such a value from any command labelled R and bind it to the variable x (R?x).

The source language static semantics

In this language the static semantics is very simple, the only constraint applied is that of non-interference through state variables as described in chapter 3 section 1.2. We will restate here the relevant portions of said definition for ease of reference. First we need to restate some auxilliary definitions:

Definition 5.1

In order to characterise the subsets of Vars which may influence (in being read) or be influenced by (in being written to) some command we introduce the following functions RV, WV, and FV from Com to the powerset of Vars, and define them in the usual tabular fashion:

RV(c) - the set of variables c may read from.

WV(c) - the set of variables that c may write to.

FV(c) = RV(c)+WV(c) - the set of free variables of c.

	x:=e;c	S!e;c	R?x;c
RV	FV(e)+RV(c)	FV(e)+RV(c)	RV(c)
WV	{x}+WV(c)	WV(c)	{x}+WV(c)

and of course RV(done) = WV(done) = ϕ.

□

Now we can go about defining non-interference as before,i.e.

Definition 5.2

For c_1, c_2 ε Com define

non-inter(c_1,c_2) if $FV(c_1) \wedge WV(c_2) = FV(c_2) \wedge WV(c_1) = \phi$

Thus if c_2 can influence variable x (i.e. $x \varepsilon WV(c_2)$) then it can be neither read nor written to by c_1 (i.e. $x \notin FV(c_1)$), and vice versa. However, c_1 and c_2 may read from the same variable if neither can alter it.

□

Thus we arrive, finally, at the single simple restriction on programs, namely that no two labelled commands in a program may interfere with each other through the medium of the shared state. Put more formally:

$$\frac{\forall\ i,j \in \{1,..,n\},\ i \neq j.\ \text{non-inter}(c_i,c_j)}{\vdash R_1::c_1\ ||...||\ R_n::c_n}$$

where $\vdash$ is the expression of syntactic validity.

The source language operational semantics

The next thing to do is introduce the semantic notions of configurations and transitions. Now seems as good a time as any to christen the language so lets call it ... 'Source'. As before

the Source language semantics is given by a labelled transition system. This system will more or less coincide with that given in chapter 2 on this restricted subclass of CSP programs.

Definition 5.3

As in chapter 3 a configuration is a piece of program text together with a binding of values to variables, i.e. a state

$$config(Source) = \{\langle p,s\rangle : \vdash p\}$$

We will define a non-empty set of terminals for use later in defining success

$$term(Source) = \{\langle R_1::c_1 || \ldots || R_n::c_n , s\rangle : c_1 = \ldots = c_n = done\}$$

Thus every command in a terminal configuration cannot, and does not wish to, perform any (more) actions.

Now we almost have a labelled transition system, the next step is to define the transition labels,

$$labels(Source) = \{\varepsilon, R::S!v, S::R?v\}$$

All that remains now is to define trans(Source). The definition of trans(Source) is as usual by a form of structural induction on the program syntax. Thus, in this case, we are in fact defining a family of transition relations over a number of transition systems since, for instance, $\langle x:=e,s\rangle \notin config(Source)$. We will finesse this point, dropping all subscripts as usual, in the belief that the reader can make the necessary distinctions.

Atomic commands

1. $$\frac{eval(e,s) = v}{\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle done, s[v/x]\rangle}$$

$$2.\quad \frac{eval(e,s) = v}{\langle S!e,s\rangle \xrightarrow{S!v} \langle done,s\rangle}$$

$$3.\quad \langle R?x,s\rangle \xrightarrow{R?v} \langle done,s[v/x]\rangle$$

Commands

The configuration <done,s> has no transitions.

$$4.\quad \frac{\langle ac,s\rangle \xrightarrow{a} \langle done,s'\rangle}{\langle ac;c,s\rangle \xrightarrow{a} \langle c,s'\rangle}$$

Programs

$$5.\quad \frac{\langle c,s\rangle \xrightarrow{a} \langle c',s'\rangle}{\langle R::c,s\rangle \xrightarrow{R::a} \langle R::c',s'\rangle}$$

$$6.\quad \frac{\langle p_1,s\rangle \xrightarrow{a} \langle p_1',s'\rangle}{\langle p_1||p_2,s\rangle \xrightarrow{a} \langle p_1'||p_2,s'\rangle,\ \langle p_2||p_1,s\rangle \xrightarrow{a} \langle p_2||p_1',s'\rangle}$$

$$7.\quad \frac{\langle p_i,s\rangle \xrightarrow{R::S!v} \langle p_i',s\rangle,\ \langle p_j,s\rangle \xrightarrow{S::R?v} \langle p_j',s'\rangle,\ \{i,j\}=\{1,2\}}{\langle p_1||p_2,s\rangle \xrightarrow{\varepsilon} \langle p_1'||p_2',s'\rangle,\ \langle p_2||p_1,s\rangle \xrightarrow{\varepsilon} \langle p_2'||p_1',s'\rangle}$$

□

It should be noted that we can state equivalent forms for 5, 6, and 7 in the $p_1||...||p_n$ notation.

5.1.2. Translation motivation

We are aiming to construct a translation from the language Source to some as yet undefined language Target, wherein the paradigm of handshake communication is replaced by a mechanism of a rather different form. We will see that a natural and simple translation can be constructed, along with a language Target to support it,

that performs this function. Also it will be a complete-implementation yet present severe difficulties to the other theories. Thus we now proceed to motivate the translation and Target language, and later describe why they have the properties we require.

We start with the very simplest case of communication between just two processes, that is there are only two processes and they both know it. The basic paradigm of the translation is to model handshake communication by placing/removing messages in/from a common area, i.e: communication will be conducted through shared variables. Consider a source configuration

$$\langle R::S!e;c_1 \;||\; S::R?x;c_2 \;,\; s\rangle$$

How do we model the handshake communication that the semantics of Source calls for? Suppose in our common area we have two 'trays', in analogy with the in/out trays on many peoples desks. Call the in tray 'send' and the out tray 'sent'. The communication proceeds as follows in stages 1 to 5:

1. Both trays are empty.

2. R places the value of e (evaluated in the state s) into the send tray, and waits for something to appear in the sent tray.

3. Meanwhile, S has been waiting for a value to appear in the send tray. On the appearence of the value placed there by R it is seized by S and stored in the variable x. Now both trays are empty again.

4. S now acknowledges its receipt of the value by placing some token, it doesn't matter what, in the sent tray and is then

free to go about its other business, i.e. c_2.

5. Still vigilantly waiting, R spots the token in the sent tray, removes it and, taking that to signal completion of the communication, continues on its way. Both trays are empty yet again, ready for the next communication.

Before we extend this method there are two points which are worth noting and which will remain valid over subsequent developments,

1. Communications are no longer 'instantaneous', they are extended over a number of phases and the particular phase occuring in some configuration is determined by both the contents of the trays and the program text. However, there is an invariant relationship, which we will describe later, between these entities which will be crucial to the proof of complete-implementation.

2. In extending a communication event over a number of phases we have introduced a slight asymmetry between senders and receivers, namely that senders start first and finish last. This 'starting first' will, in the following refinement, be the undoing of any attempt at a bisimulation approach to correctness.

Having demonstrated the simplest case we go on to extend the technique to many communicating processes, some of which have the same name. To cope with many processes we index the trays by the names of the sender and receiver, thus

send becomes (R,S,send)

to contain the value R wishes to send to S, and

sent becomes (R,S,sent)

to contain the acknowledgement that S wishes to send to R.

Hence we have a number of trays indexed by

Proc x Proc x {send,sent}

each triple being ('sender','receiver',send) or ('sender','receiver',sent).

If now we were to allow each tray to contain an unordered set of values then we would have two problems;

1. Values are associated with <u>particular</u> processes; consider the following configuration:

   <R::S!e_1;x:=e_3 || R::S!e_2 || S::R?y , s>

   A possible computation using trays as described is:

   1. The left R puts the value of e_1 in tray (R,S,send) and waits.

   2. The right R puts the value of e_2 in tray (R,S,send) and waits.

   3. S takes the value of e_2 from the tray (R,S,send) and puts it in variable y.

   4. S puts a token into the tray (R,S,sent) and finishes.

   5. <u>Now</u> the <u>left</u> R takes the token, leaving (R,S,sent) empty, and finishes by assigning the value of e_3 to x.

   This is unacceptable since we now have a configuration where y has the value of e_2, implying S communicated with the right(most) R, but x has the value of e_3, implying that S

communicated with the left(most) R.

The solution is to have each R provide a unique identifying token with the value it places in the send tray, it is this token which is returned by S in the sent tray and each sending process can only retrieve the token it sent with its value.

2. If the set is unordered we would still be left with the problem of choosing among the alternatives. it would be nicer to have some deterministic mechanism make the choice and to this end we will demand that each tray forms a queue of its contents, i.e. a first-in-first-out (FIFO) list. (Unfortunately the analogy with trays falls down a bit here since an office tray functions as a stack!). Thus senders of the same name to any receiver of a particular name must queue up, and their acknowledgements are similarly queued out.

   The translation would still be correct in the sense of 'the testing theory if we removed this determinism but besides being quite a natural addition it will turn out that this extra determinism will preclude any possiblity of applying the bisimulation theory in an acceptable way.

Now that we have informally motivated the translation, and thereby the Target language, we next formalise the Target language and then go on to consider the notion of experimentation to be employed.

5.1.3. The Target language

As mentioned earlier the Target language will share parameters, both syntactic (Proc, Expr, Vars) and semantic (States, eval etc.), with the Source language. The essential difference is in the mode of communication, that is shared variables rather than handshaking. This difference is achieved in two changes to Source, the introduction of a communication state and the substitution (for the handshake commands R?x, S!e) of atomic commands to manipulate that state. In order to define communication states we need first to define queues properly

Definition 5.4

A queue, q, over some set A of queue elements is an ordered finite submultiset of A

$$q = \langle a_1,\ldots,a_n\rangle, \quad n \geq 0$$

The set of all such queues is written

Queues(A)

We need some notation for describing queues; suppose $q = \langle a_1,\ldots,a_n\rangle$, then

1. mset(q) is the corresponding unordered multiset:

$$mset(q) = [a_1,\ldots,a_n]$$

set(q) is the corresponding unordered set:

$$set(q) = \{a_1,\ldots,a_n\}$$

If there are no repetitions in the multiset mset(q) then we write norep(q) .

2. empty is the empty queue (in the case n = 0):

$$q = empty \qquad \text{if } set(q) = \phi$$

3. $$q = a_1.q' \qquad \text{if } n \geq 1,\ q' = \langle a_2,\ldots,a_n\rangle$$

i.e. a_1 is at the front of the queue.

4. $q = q'.a_n$ if $n \geq 1$, $q' = \langle a_1,\ldots,a_n \rangle$

i.e. a_n is at the end of the queue.

As usual with queues it is expected that removals from a queue are done at the front, and additions at the rear. It is only under such conditions of use that they can (meaningfully) be called queues.

□

We now define communication states, as semantic entities like normal states, in accord with the earlier informal motivation of trays as queues.

Definition 5.5

A communication state Q is a triple of functions

f: Proc x Proc x {send} -> Queues(Nat x Expr)

g: Proc x Proc x {sent} -> Queues(Nat)

h: Nat -> Nat

The intent is that the tokens identifying processes uniquely are natural numbers (we will not bother to distinguish numbers from numerals, and let t (for token) range over Nat). Since the domains of f, g, and h are disjoint, and their elements easily recognised, we just write Q for each of f, g, and h. Thus

Q(R,S,send)- is a queue of token-expression pairs, the 'send' tray. We employ expressions here rather than values to facilitate certain aspects of the correctness proof. A translation which placed values rather than expressions on the queues would be correct also.

Q(R,S,sent)- is a queue of tokens (natural numbers), the 'sent' tray.

Q(t)- is a token t'. This is where a receiver whose token is t remembers that the corresponding sender is t' and may be thought of as representing a set of single element 'pending' trays, one for each process (see the dynamic semantics of the atomic command 'rec' later on).

□

For the purposes of the translation we will need a particular initial communication state which says that no communications are taking place.

Definition 5.6

The communication state <u>initQ</u> is defined by

∀ R, S. initQ(R,S,send) = empty, initQ(R,S,sent) = empty.

∀ t ε Nat. initQ(t) = 0.

(It is intended that no process will be assigned the identifying token 0).

□

The Target language is meant to be as close as possible to the Source language except for the change in communication mode. Thus its description will closely parallel that of Source and it will be fully explained only at points of departure.

The Target language syntax

There are three major syntactic categories as before

<u>Acom2</u> - a set of 'atomic commands' ranged over by **ar**, defined by

ar ::= x:=e | off(qvd,<t,e>) | rec(qvd,<t',x>) |

akn(qvt,<t'>) | wait(qvt,<t>)

where qvd ranges over Proc x Proc x {send} and qvt ranges over Proc x Proc x {sent}

For the intended interpretation of these syntactic entities the puzzled reader should skip to the section on dynamic semantics.

Com2 - a set of 'commands' ranged over by r and defined by

r ::= ar;r | done

Prog2 - a set of 'programs' ranged over by pr and defined by

pr ::= R::r | pr_1||pr_2

As before, in Prog, some descriptions of elements of Prog2 may be written as if Prog2 was defined by

pr ::= R_1::r_1 ||...|| R_n::r_n

□

The Target language static semantics

The only syntactic restriction applied in the Target language is the direct analogue of that in Source, viz. non-interference. Thus, in the interests of brevity, we merely redefine the sets of variables written or read:

Definition 5.7

	off(qvd,<t,e>);r	rec(qvd,<t',x>);r
RV	FV(e)+RV(r)	RV(r)
WV	WV(r)	{x}+WV(r)

	akn(qvt,<t'>);r	wait(qvt,<t>);r
RV	RV(r)	RV(r)
WV	WV(r)	WV(r)

Finally FV(r) = RV(r)+WV(r).

□

The Target language dynamic semantics

There are quite a few changes here so we'll go a bit slower; we first define the transition system Target

Definition 5.8

A configuration is now a **triple**; a piece of program text, a communication state, and a normal state:

$$config(Target) = \{\langle pr,Q,s\rangle : \vdash pr\}$$

As in Source, the terminal configurations are those in which all component commands are exhausted (we make no conditions on the communication state, but for exhausted translations we will be able to prove it is initQ).

$$term(Target) = \{\langle R_1::r_1 \;||...||\; R_n::r_n,Q,s\rangle : r_1 = ... = r_n = done\}$$

The labels which can appear on arrows are, of course, heavily depleted; communications pass 'silently' through the communication state,

$$labels(Target) = \{\varepsilon\}$$

The definition of trans(Target) (i.e. ->) proceeds in the same muddled fashion as employed in Source, confusing different ternary relations in the same -> symbol,

Atomic commands

$$1.\quad \frac{eval(e,s) = v}{\langle x:=e,s\rangle \xrightarrow{\varepsilon} \langle done, s[v/x]\rangle}$$

2. $\langle off(qvd,\langle t,e\rangle), Q[q/qvd], s\rangle \xrightarrow{\varepsilon} \langle done, Q[q.\langle t,e\rangle/qvd], s\rangle$

Thus the 'offer' command 'off((R,S,send), <t,e>)' will place the expression e, along with a token t identifying the process containing this command, on the rear of the queue denoted by (R,S,send).

3.

$$\frac{eval(e,s) = v}{\langle rec(qvd,\langle t',x\rangle),Q[\langle t,e\rangle.q/qvd],s\rangle \xrightarrow{\varepsilon} \langle done,Q[q/qvd,t/t'],s[v/x]\rangle}$$

The 'receive' command makes three changes if qvd in the current communication state denotes a non-empty queue:

1. The element <t,e> is removed from the queue.

2. The value v (= eval(e,s)) is assigned to x. Note this introduces a slight subtlety into the interpretation of RV(off(qvd,<t,e>);r) since it is not the sender who evaluates the expression but rather the receiver.

3. The token t of the sender is associated, in the communication state, with the token t' of the receiver. This is how the receiver remembers the identity of the sender.

4.

$$\langle akn(qvt,\langle t'\rangle),Q[q/qvt,t/t'],s\rangle \xrightarrow{\varepsilon} \langle done,Q[q.\langle t\rangle/qvt, 0/t'],s\rangle$$

The command to 'acknowledge' has two effects

1. The process containing this command will be identified by the token t'; akn will look up, in the communication state, the token t of its associated sender, and place this on the back end of the queue denoted by qvt.

2. The association of t with t' is undone. Though this is not strictly necessary it is helpful for future definitions and proofs.

5. $\langle wait(qvt,\langle t\rangle), Q[\langle t\rangle.q/qvt], s\rangle \xrightarrow{\varepsilon} \langle done, Q[q/qvt], s\rangle$

On the appearence of the correct token, t, at the front of the queue denoted by qvt the 'wait' command can terminate, after removing t from the queue.

Commands

6. $$\frac{\langle ar, Q, s\rangle \xrightarrow{\varepsilon} \langle done, Q', s'\rangle}{\langle ar;r, Q, s\rangle \xrightarrow{\varepsilon} \langle r, Q', s'\rangle}$$

Programs

7. $$\frac{\langle r, Q, s\rangle \xrightarrow{\varepsilon} \langle r', Q', s'\rangle}{\langle R::r, Q, s\rangle \xrightarrow{\varepsilon} \langle R::r', Q', s'\rangle}$$

8. $$\frac{\langle pr_1, Q, s\rangle \xrightarrow{\varepsilon} \langle pr_1', Q', s'\rangle}{\langle pr_1||pr_2, Q, s\rangle \xrightarrow{\varepsilon} \langle pr_1'||pr_2, Q', s'\rangle}$$

9. $$\frac{\langle pr_2, Q, s\rangle \xrightarrow{\varepsilon} \langle pr_2', Q', s'\rangle}{\langle pr_1||pr_2', Q, s\rangle \xrightarrow{\varepsilon} \langle pr_1||pr_2, Q', s'\rangle}$$

□

5.1.4. The translation $[\![.]\!]_1$

In chapter 4 section 2.4 a translation was defined as a function between **experiments** so it may seem a little premature to be describing the translation before the experiment systems, but for **this** example the old notion of translation (given in chapter 2) as a function from source language configurations to target language configurations, extends smoothly to experiments. Thus we will give an 'old-style' translation now, and extend it to cover experiments later.

By now the translation must be quite obvious to the reader; however, it still remains to show how tokens are assigned to processes and thence to their constituent atomic commands. The technique, as in chapter 3, is to subscript the usual context-free translation function with some context-sensitive information, in this case the process label and identifying token of the process that will contain the translated form.

As usual the semantic translation between configurations is generated by a syntactic translation. Strictly speaking we are employing a number of translations differing in argument type, subscripts etc., but again we finesse these minor details.

Definition 5.9

Commands

$$[\![\text{done}]\!]_{(R,t)} = \text{done}$$

$$[\![x:=e;c]\!]_{(R,t)} = x:=e;[\![c]\!]_{(R,t)}$$

$[\![S!e;c]\!]_{(R,t)}$ = off((R,S,send), <t,e>);
wait((R,S,sent), <t>);
$[\![c]\!]_{(R,t)}$

$[\![R?x;c]\!]_{(R,t')}$ = rec((R,S,send), <t',x>);
akn((R,S,sent), <t'>);
$[\![c]\!]_{(R,t')}$

Programs

$$[\![R_1::c_1 \;||\ldots||\; R_n::c_n]\!] = R_1::[\![c_1]\!]_{(R_1,1)} \;||\ldots||\; R_n::[\![c_n]\!]_{(Rn,n)}$$

Now, the generated semantic translation between configurations is given by

Configurations

$[\![<p,s>]\!]_1$ = <$[\![p]\!]$, initQ, s>

□

5.1.5. $[\![.]\!]_2$ and the experiment systems SEXP, TEXP

Having described the Source and Target languages and their associated transition systems we need a notion of experimentation on configurations from those systems in order that we can apply the testing theory of translation correctness developed in chapter 4. Fortunately the notion of experimentation will be the same for both systems, so we will describe the construction of SEXP in detail and mention TEXP only at points where the corresponding construction of TEXP is unclear. SEXP (TEXP) is, of course, the Source (Target) experimentation system.

An experiment is as usual a test-process pair, the test comes in two pieces;

Dynamic communication testing, as described in chapter 4 section 2, wherein another process (as part of the test) communicates with the process being tested in a possibly extended dialogue. If the pair reach some 'agreeable' joint configuration then the second part of the test is applied.

Final state testing, an 'agreeable' configuration is successful if its final component state (i.e. binding of values to variables) is a member of some set of states P which, together with the testing process, completely determines the test.

Definition 5.10

We first define the processes to be tested:

OBJ = config(Source), ranged over by **obj**.

Each test is a pair; a piece of program text to do the dynamic testing plus a subset of States for the final test:

TST = {(p,P): p ε Prog, P $\underline{C}$ States}, ranged over by **tst**

The application of a test to a process involves merging the two program texts, to be run as a single program:

$tst[obj] = (\langle p_1 || p_2, s\rangle, P)$

if $tst = (p_1, P)$, $obj = \langle p_2, s\rangle$, $\vdash p_1 || p_2$

The merged program texts must form a valid (i.e. $\vdash$ from the static semantics of Source) program, otherwise the test might interfere in the internal workings of the tested process by altering some (shared) variable. It might also be that the test could gain privileged knowledge by reading the value of some variable altered by the tested process. While there is no reason

that such tests shouldn't be applied in such a way, it is not the intended form of testing for our example and is therefore excluded.

Experiment configurations are just applications of tests to processes:

E = {tst[obj]: tst ε TST, obj ε OBJ}, ranged over by e.

Success has two components, the experiment must have finished and it must have an acceptable final state:

success((<p,s>, P)) iff <p,s> ε term(Source), s ε P

It remains to define the transition relation between experiments. The set P takes no part in these transitions, it's just "there for the ride".

$$\frac{\langle p,s\rangle \xrightarrow{\underline{\varepsilon}} \langle p',s'\rangle}{(\langle p,s\rangle,\ P) \rightarrow (\langle p',s'\rangle,\ P)}$$

where $\xrightarrow{\underline{\varepsilon}}$ $\underline{C}$ trans(Source). Thus experiments will proceed with no outside interference by any observer in any way. Finally,

SEXP = <E, ->, success>

□

The definition of TEXP is analogous to that for SEXP, thus here we will give the bare definitions with little justification.

Definition 5.11

1. TOBJ = config(Target), ranged over by **tobj**
2. TTST = {(pr,P): pr ε Prog2, P $\underline{C}$ States}, ranged over by **ttst**
3. ttst[tobj] = (<pr_1||pr_2, Q, s>, P)

   if ttst = (pr_1,P), tobj = <pr_2, Q, s>, $\vdash$ pr_1||pr_2

4. F = {ttst[tobj]: ttst[tobj] is defined}, ranged over by f

5. tsuccess((<pr, Q, s>, P)) iff <pr,Q,s> ε term(Target), s ε P

Note we will generally write tsuccess as 'success' where no confusion can arise.

$$\frac{\langle pr, Q, s\rangle \xrightarrow{\varepsilon} \langle pr', Q', s'\rangle}{(\langle pr, Q, s\rangle, P) \rightarrow (\langle pr', Q', s'\rangle, P)}$$

And so to ...

TEXP = <F, ->, success>

□

It is now a simple matter to give the extended translation, i.e. a total function from source experiments to target experiments.

Definition 5.12

$$[\![(\langle p,s\rangle, P)]\!]_2 = ([\![\langle p,s\rangle]\!], P)$$

□

Thus the state part of the test is passed over verbatim and the remainder of the experiment is translated as a configuration of Source, so that the process part of the test is translated into a form compatible with the translation of the process under test.

To emphasise the point once more we can say that we have in fact 'translated' the test as well as the process, and in considering translation correctness we will compare the results of applying test to process with the results of applying the 'translated' test to the translated process.

5.2. Proving correctness of the translation $[\![.]\!]_2$

This section has two main aims,

1. To demonstrate that all of the translation theories discussed so far (i.e. the bisimulation theory of chapter 2, the 'correctness' theory of Li [Li 83], the testing theory of chapter 4, and the theory proposed in [Astesiano and Zucca 82]) only the testing theory can cope with this example translation.

2. To provide a set of four conditions which are sufficient to guarantee a translation between experiment systems is a complete implementation, given two demands on the nature of the systems. We then establish the truth of these four conditions for the example translation, and finally go on to discuss how one of the two demands could be relaxed by the addition of a fifth condition.

It will be helpful to introduce now the demands on the nature of the experiment systems, taken together they will amount to requiring that the computation set of every experiment (in either system) contains only finite computations, and every computation ending in an unsuccessful configuration contains no successful configurations. That is to say; every computation is finite and the computation is successful if and only if the final (or tip) configuration is successful.

Definition 5.13

An experiment system EXP = <E, ->, success> is <u>computationally finite</u> if $\forall\ e \in E.\ \langle c \rangle_K \in Comp(e)$ implies K is finite.

□

Definition 5.14

An experiment system EXP = <E, ->, success> is <u>success terminating</u> if $\forall$ e ε E. success(e) implies e $-\not\rightarrow$.

□

Proposition 5.1

SEXP and TEXP are both computationally finite and success-terminating.

□

5.2.1. Why the bisimulation approach won't work

In the previous section we defined two translations

$[\![.]\!]_1$: config(Source) -> config(Target)

$[\![.]\!]_2$: config(SEXP) -> config(TEXP)

We now examine the applicability of each translation theory to $[\![.]\!]_1$ and $[\![.]\!]_2$, primarily from the context of a bisimulation approach.

$[\![.]\!]_1$ - Clearly the bisimulation theory and the equivalence theory of Astesiano and Zucca must deny $[\![.]\!]_1$ as a 'correct' translation because

R::S!v $\not\in$ labels(Target)

but

<R::S!e, s> $\xrightarrow{\underline{R::S!v}}$ <done, s> if eval(e,s) = v.

In effect all the 'observable' actions on the arrow for the Source system have been removed. Any attempt to 'observe' an action in the Target system involves manipulation of a communication state, which is, essentially, how the testing approach works for $[\![.]\!]_2$. The testing theory of course is not applicable to $[\![.]\!]_1$ directly since Source and Target are not experiment systems.

Li's correctness theory fails immediately for a similar reason. Part of Li's definition of correctness [Li 83 page 196] was a condition

A1. $\forall$ r. tr(R(r)) = R(tr(r))

where tr is a semantic translation (extended here on the left hand side to cover sets of configurations), r is a source configuration and the 'result set' R(r) is defined for source (target) configurations as ([Li 83 Definition 5.1 page 195]):

$R(r) = \{r' : \nexists\ w.\ r \xrightarrow{w} r',\ r' \in T\}$

where T is the set of terminal source (target) configurations. Thus A1 essentially says that the possible successfully terminated configurations reachable from the translation of configuration r are exactly the translations of successfully terminated configurations reachable from r.

We can see that A1 does not hold for tr = $[\![.]\!]_1$; for example, take

$r = \langle R::Sle, s\rangle \in config(Source)$

then, by Li's definitions we have

$R(r) = \{\langle done, s\rangle\}$

Now,

$tr(r) = \langle R::off(\_);wait(\_), initQ, s\rangle$

and

$R(tr(r)) = \phi$

since $\forall$ Q', s'

$\langle R::wait(\_), Q', s'\rangle \notin term(Target)$

What has happened?. The problem is caused by Li's heavy emphasis on 'final result' states, to the detriment of notions about observation. Correct translations, essentially, assume

configurations interact with their environment through the labels on their transitions, which for this example translation is not so; communication is affected by altering a shared communication state.

Li then goes on, in the definition of the sets R and K (a similar definition to R but for deadlocked states) to ignore action labels altogether when gathering result configurations and deadlocked configurations. There is no concern for how a configuration may be 'guided', by an observer specifying a particular sequence of action labels, into one set or the other.

$[\![.]\!]_2$ - Since experiment systems were constructed specifically for the testing theory, applying any of the other theories to a translation between them is is, at least, a partial acceptance of that paradigm. The acceptance is essentially that configurations from a transition system should be observed (tested) in a manner relevant to that system . What might not be accepted is the use of must and may in defining correctness. We will show that in eschewing the must/may approach, the bisimulation (and also the equivalence theory of [Astesiano and Zucca 82]) runs into serious difficulties.

Li's theory fares rather better on $[\![.]\!]_2$ since it always did ignore transition labels and there are none in experiment systems. However, the 'adequacy' conditions are very much concerned with action labels and thus do not present a viable proof technique in this case. As to the exact nature of any result proven about $[\![.]\!]_2$ we merely note that, under the testing interpretation of $[\![.]\!]_2$, the conditions for correctness are reasonable if rather stringent.

(Note complete-implementation does not imply A1,...,A3). Thus 'correctness' could supplant 'complete-implementation' in this case, but in general would probably be too restrictive, particularly if the translation was only an 'implementation' or a different notion of test was required.

Before we go into any further details it must be appreciated that it is very difficult to completely discredit the bisimulation approach, and probably any other, since it may give some kind of interesting positive result when applied in an unusual way. That is of course exactly what we are trying to do in applying it to $[\![.]\!]_2$ rather than $[\![.]\!]_1$. We will, however, endeavour to demonstrate why many intuitively reasonable applications of the bisimulation theory must fail, by attempting to prove $[\![.]\!]_2$ is P-good for a 'reasonable' P (and failing). We start by choosing P.

It should be noted that now we are translating between experiment systems the only action label is ε so most of the work must be done by P. Probably the weakest 'reasonable' choice is to compare states at the tips of the, in this case, finite computation tree. Firstly we need an auxilliary definition to gather the possible final states of an experiment; let => = ->*,

Definition 5.15

Let e ε exp(SEXP), f ε exp(TEXP)

resultset(e) = {s: e => (<p,s>, P) -/>}

resultset(f) = {s: f => (<pr, Q, s>, P) -/>}

□

Now, we will just demand that any experiments which cannot proceed

and which are paired in the bisimulation must have the same state component. (The example to follow will have no deadlocked states so confining comparisons to pairs of terminal states (say) will not help).

Definition 5.16

(<p,s>, P) **result** (<pr, Q, s'>, P)

iff

(<p,s> -/>, <pr, Q, s'> -/>) implies s = s'

□

The following proposition establishes a crucial property of any result-bisimulation, namely that any experiments paired in the bisimulation must have the same set of final states.

Proposition 5.2

1. e => e' implies resultset(e') $\underline{C}$ resultset(e)

   f => f' implies resultset(f') $\underline{C}$ resultset(f)

2. Suppose e R f, R a result-bisimulation (so R $\underline{C}$ exp(SEXP) x exp(TEXP)), then resultset(e) = resultset(f).

Proof

1. Suppose e => e' and

   e' => e" -/>

   then

   e => e" -/>

   so resultset(e') $\underline{C}$ resultset(e)

   Similarly for f.

2. Suppose

   e => e' -/>

   then

$\exists$ f'. f => f', e' R f'

Now, since e' -/>

$\forall$ f". f' => f" -/> implies e' R f'

and, because all computations are finite,

$\exists$ f". f' => f" -/>

Thus,

$\exists$ f". f => f" -/>, e' R f"

so by Definition 5.15, Definition 5.16

resultset(e) $\underline{C}$ resultset(f)

Similarly resultset(f) $\underline{C}$ resultset(e)

□

In the following example we will demonstrate that $[\![.]\!]_2$ is not a result-good translation by showing that a translated experiment can reach a configuration which cannot be paired, in a result-bisimulation, with any configuration attainable by the original experiment.

Example

The experiment we wish to translate is composed of 5 processes in parallel, two senders (named R) and three receivers (named S). Any complete computation of this experiment will determine a pairing of senders with receivers which will be reflected in the final state.

Notation States will be represented as

$s = \langle v_1, v_2, v_3 \rangle$ if eval(x,s) = v_1, eval(y,s) = v_2, eval(z,s) = v_3

□

So, take

e = (<R::S14 || R::S15 || S::R?x || S::R?y || S::R?z,
<0,0,0>>,
ϕ)

The set of 'successful' final states can be ϕ because we are not going to use it here, similarly the distinction between test and process is ignored. The initial state is chosen to show up the final assignments, since all assignments are non-zero. Now, by consideration of the operational semantics it is easily seen that

$[\![e]\!]_2$ => (<R::wait(_) || R::wait(_) || _,
initQ[<1,4>.<2,5>/(R,S,send)]
<0,0,0>>,
ϕ)
= f

Thus, in effect, f has contracted to choose the receiver for the value 4 **before** choosing the receiver for the value 5. The experiment e cannot express such a contract and so no pairing can be found for f in a result-bisimulation containing $(e,[\![e]\!]_2)$ as we shall show.

We have

$[\![e]\!]_2$ => f

thus if $[\![.]\!]_2$ is result-good, with result-bisimulation R, then

$\exists$ e'. e => e', e' R f

(Note we are using => rather than -> as the transition relation).

The example is simple enough that we can search for e' by simple case analysis

<u>case</u> e = e': but

e -> (<R::S!4 || R::done || S::done || _,

<5,0,0>>,

ϕ)

= e"

i.e. e can choose the receiver for 5 first. Thus

resultset(e") = {<5,0,4>, <5,4,0>}

We now show f has no 'corresponding' derivative; suppose

f => f', e" R f'

then either

f = f': but then

<4,0,5> ε resultset(f')

and

<4,0,5> ∉ resultset(e")

which contradicts Proposition 5.2(2)

or, since the next move of f must be to pick the receiver for 4, one of the following must hold by Proposition 5.2(1)

1. resultset(f') $\underline{\subseteq}$ {<4,0,5>, <4,5,0>} = set_1

2. resultset(f') $\underline{\subseteq}$ {<0,4,5>, <5,4,0>} = set_2

3. resultset(f') $\underline{\subseteq}$ {<0,5,4>, <5,0,4>} = set_3

depending on which of x (set_1), y (set_2), or z (set_3) was chosen. However,

∀ i ε {1,2,3}. resultset(e") $\underline{\not\subseteq}$ set_i

thus by Proposition 5.2 (2) there can be no f'.

In effect this demonstrates that, in some sense, f has less choice than e (or has made more decisions).

<u>case</u> e' = (<R::done || R::S!5 || S::done || _,

<4,0,0>>,

ϕ]

but then

resultset(e') = {<4,0,5>, <4,5,0>}

whereas

<5,0,4> ε resultset(f)

so by Proposition 5.2(1,2) neither e' nor any of its derivatives can be paired with f. Thus in this case f has more choice than e', and this is true of the other remaining cases which we will not explore.

We have thus demonstrated that f cannot be paired in a result-bisimulation, and thus $[\![.]\!]_2$ cannot be result-good.

5.2.2. The four conditions

The technique we will employ to prove $[\![.]\!]_2$ is a complete implementation relies on characterising the tips (final configurations) of the computations for translated experiments as elements of a set $[\![e]$ (for some e dependent on the particular tip). The set $[\![e]$ comprises those configurations f which can be obtained from $[\![e]\!]$ by putting all possible communication offers in the appropriate send-queue. This enables, from conditions c2 and c3 to be defined, the construction of a correspondence between tips of the source computations and tips of the target computations. Since we are not interested in choice points along the way, the downfall of the bisimulation theory, we needn't consider all possible forms the translated experiment might attain; just the forms taken at the tips. This will give condition c1. Condition c4 will establish criteria of success for

these tip forms.

At the end of this chapter we will discuss how these conditions may be generalised and extended to apply to experiment systems which are not computationally finite.

Theorem 5.1

Suppose

1. $S = \langle E_s, \rightarrow_s, success_s \rangle$, $T = \langle E_t, \rightarrow_t, success_t \rangle$ are computationally finite success terminating experiment systems, e ranges over E_s and f ranges over E_t. (Subscripts will be dropped from now on where unnecessary).
2. $[\![.]\!]: E_s \rightarrow E_t$, $[\![.)\!]: E_s \rightarrow P(E_t)$

then

c1. $\forall$ e, e'. e => e' iff $\exists$ f ε $[\![$e'$)\!]$. $[\![$e$]\!]$ => f

c2. $\forall$ e, f. $[\![$e$]\!]$ => f $\not\rightarrow$ implies $\exists$ e'. f ε $[\![$e'$)\!]$

c3. $\forall$ e. $\forall$ f ε $[\![$e$)\!]$. e -> iff f ->

c4. $\forall$ e. $\forall$ f ε $[\![$e$)\!]$. success(e) iff success(f)

implies

$\forall$ e. $[\![$e$]\!]$ <u>completely-implements</u> e

□

If we were only interested in implementation then it would suffice to prove,

a) The 'if' part of c1

b) c2

c) The 'only if' part of c3

d) c4

It will turn out that most of the work in the proof of complete implementation is concerned with establishing a) and b)

Proof of Theorem 5.1;

1. (e must implies [[e]] must): Suppose e must and take any computation (which will be finite) of the translation

$$[[e]] \Rightarrow f \not\rightarrow$$

then by c2

$$\exists\, e'.\ f \in [[e']$$

so by c1

$$e \Rightarrow e'$$

and by c3

$$e' \not\rightarrow$$

Thus, since S is success terminating and e must (by hypothesis),

$$\text{success}(e')$$

and hence by c4

$$\text{success}(f).$$

2. (e may implies [[e]] may): Suppose e may, then

$$\exists\, e'.\ e \Rightarrow e',\ \text{success}(e')$$

Thus by c1

$$\exists\, f \in [[e']).\ [[e]] \Rightarrow f$$

and by c4

$$\text{success}(f)$$

3. ([[e]] must implies e must): Suppose [[e]] must; take any computation (finite by supposition 1))

$$e \Rightarrow e' \not\rightarrow$$

then by c1

$$\exists\, f \in [[e']).\ [[e]] \Rightarrow f$$

and by c3

$$f \not\rightarrow$$

Thus, since [[e]] must,

success(f)

and hence by c4

success(e')

4. ([[e]] may implies e may): Suppose [[e]] may, then since T is success terminating

[[e]] => f -/>, success(f)

By c2

∃ e'. f ε [[e')

so by c1

e => e'

and c4 gives

success(e')

□

We now pause to outline the method by which we will prove conditions c1 to c4. The proof rests on three crucial definitions

[[e) - is the set of target configurations which can be obtained from [[e]] by 'running' through one step all and only (and in any order) those labelled commands in [[e]] whose first step is to place a communication offer in a send queue. c3 and c4 follow immediately from this definition.

cons(f) - is true if the target experiment f has a program and communication state which are consistent in the sense that f can be regarded as '[[e]] doing something' for some e.

inv(f) - if f can be regarded as '[[e]] doing something' then inv(f) = e; basically inv is an inverse translation function defined on consistent f.

The following properties of these definitions will hold

1. ∀ e. (cons([[e]]), ∀ f ε [[e]]. cons(f))

2. (cons(f), f => f') implies cons(f')

3. cons(f) implies inv(f) is defined.

4. ∀ e. (f ε [[e]] implies inv(f) = e, inv([[e]]) = e)

5. (cons(f), f => f') implies inv(f) => inv(f')

6. (cons(f), f -/>) implies f ε [[inv(f)]]

Using these 6 properties we can prove c2 and the 'if' part of c1 by

c1 (if): Suppose

∃ f. [[e]] => f ε [[e']]

then

cons(f), cons([[e]]) (1 and 2)

so

inv([[e]]) => inv(f) (3 and 5)

hence

e => e' (4)

c2: Suppose

[[e]] => f -/>

then

cons(f) (1)

so

f ε [[inv(f)]] (6 and 3)

We have already noted that c3 and c4 are immediate from the definition of [[.]] so all that remains is the 'only if' part of c1 which can be straightforwardly proved without recourse to the above definitions.

5.2.3. Proving the four conditions for $[\![.]\!]_2$

Before we can prove conditions c1 to c4 in this case we must define exactly what $[\![.)$ refers to. This involves a number of prior definitions which, of course, we give first.

IMAGE is the set of all possible derivatives pr ε Prog2 which may occur in a computation of some translated experiment. This will be proved later but should be intuitively acceptable anyway.

Definition 5.17

First define $INTER_t$ the intermediate syntactic forms for labelled commands with identifying token t, ranged over by $inter_t$, by

$$inter_t ::= [\![R::c]\!]_{(R,t)} \mid R::wait((R,S,sent), \langle t\rangle); [\![c]\!]_{(R,t)} \mid S::akn((R,S,sent), \langle t\rangle); [\![c]\!]_{(S,t)}$$

and define

$$INTER = \sum_{t\varepsilon Nat} INTER_t$$

Note carefully the use of subscripts <u>within</u> $inter_t$ (i.e. $inter_t$ 'knows' it is the t'th labelled command from the left).

Now define

IMAGE - ranged over by u,

$$u ::= inter_1 \;||...||\; inter_n \;,\; n \geq 1.$$

□

For each configuration occurring in a computation of a translated experiment we need to know which labelled commands are doing what.

Definition 5.18

Let $f = (\langle u, Q, s\rangle, P) \varepsilon\ exp(TEXP)$

$$u = inter_1 \,||...||\, inter_n \in IMAGE$$

then we will define the multisets (in fact they will contain no repetitions, but it helps the definitions along)

waiting(f,R,S) - the identifying tokens of all labelled commands in f which have offered to send a value from R to S, but which have not (yet) received an acknowledgment from a receiver.

akning(f,R,S) - the identifying tokens of all labelled commands in f which have just received a value as part of a communication, between some sender R and receiver S, but have not yet acknowledged the appropriate sender.

assigners(f) - the identifying tokens of all labelled commands in f which are about to execute an assignment statement.

offering(f,R,S) - the identifying tokens of all labelled commands in f which are about to offer a communication between sender R and receiver S.

by

$$waiting(f,R,S) = [t: inter_t = R::wait((R,S,sent), <t>); [\![c]\!]_{(R,t)}]$$

$$akning(f,R,S) = [t: inter_t = S::akn((R,S,sent), <t>); [\![c]\!]_{(S,t)}]$$

$$assigners(f) = [t: inter_t = R::x:=e; [\![c]\!]_{(R,t)}]$$

$$offering(f,R,S) = [t: inter_t = [\![R::S!e;c]\!]_{(R,t)}]$$

Further, we take

$$senders(f) = \sum_{R,S \in Proc} waiting(f,R,S)$$

$$receivers(f) = \sum_{R,S \in Proc} akning(f,R,S)$$

$$offerers(f) = \sum_{R,S \in Proc} offering(f,R,S)$$

□

Note all these multisets will in fact have no repetitions (i.e. they will be normal sets) but defining them as multisets will facilitate the definition of 'cons' (to follow) because multisets would show up any repetitions and part of the statement of 'cons' is that there aren't any repetitions.

We now define a relation 'run' to determine the derivatives of an experiment f given a set of process tokens identifying those labelled commands to be 'advanced one step' (in any order).

Definition 5.19

In 'do-one' we formalise the notion of executing a single labelled command determined by some token

Let $f = (\langle u,Q,s\rangle, P)$,

$u = inter_1 \,||...||\, inter_n \in IMAGE$

then define

do-one(t,f,f') if

1. $1 \leq t \leq n$,

2. $\langle inter_t,Q,s\rangle \xrightarrow{\varepsilon} \langle inter_t',Q',s'\rangle$,

3. $\forall\, i,\ 1 \leq i \leq n.$

$$inter_i'' = \begin{cases} inter_i & \text{if } i \neq t \\ inter_t' & \text{otherwise} \end{cases}$$

4. $u' = inter_1'' \,||...||\, inter_n''$,

5. $f' = (\langle u',Q',s'\rangle, P)$.

To run a set of tokens do them one at a time in any order:

run(ϕ,f,f).

run(S+{t},f,f") if

do-one(t,f,f'),

run(S,f',f").

Note S is not a multiset so each process can advance at most one step.

□

We will of course need the following:

Proposition 5.3

Suppose f = (<u,Q,s>, P),

$u = inter_1 \,||...||\, inter_n \in IMAGE$,

$S \subseteq \{1,...,n\}$

then

1. run(S,f,f') implies f => f'
2. S ≠ φ, run(S,f,f') implies f -> => f'

□

It is now quite easy to characterise the tips of computations for translated experiments as 'almost-translated' experiments, i.e. configurations reached from a translated experiment by making all possible offers of communication in some order.

Definition 5.20

$[\![e)\!) = \{f: offerers([\![e]\!]) = S, run(S,[\![e]\!],f)\}$

□

Note that although all tips are of this form, not everything of this form is a tip, e.g.

$[\![(<R::x:=e,s>, P)\!) = \{(<R::x:=e, initQ, s>, P)\}$

whose single element has a non-trivial derivation.

Proposition 5.4

1. ∀ e. ∀ f ∈ $[\![e)\!)$. $[\![e]\!]$ => f
2. Suppose f ∈ $[\![e)\!)$ for some e, f = (<_,Q,_>, _) so Q is the

communication state of f, then

1. ∀ t. Q(t) = 0

2. ∀ R, S. Q(R,S,sent) = empty

3. ∀ R, S.

   [t: ∃ e. <t,e> ∈ mset(Q(R,S,send))] = offering([[e]],R,S)

4. ∀ e. [[e] = ϕ

□

Now, the crucial element in the proof of conditions c1 to c4 is the recognition of an invariant property in all derivatives of a translated experiment. This property relates program text to communication state, saying essentially that

> If there is a sender (in the program text) waiting for acknowledgement then its unique identifying token is in one (and only one) of:
>
> 1. The appropriate send queue.
> 2. The 'pending tray' of a single receiver (and the receiver has a particular syntactic form).
> 3. The appropriate sent queue.

Given said invariant, to be called 'cons' (for 'consistent', we need 'inv' for 'inverse'), we can construct an inverse translation 'inv' so that (loosely)

f -↛>, cons(f) implies f ∈ [[inv(f)]

(which will give condition c2 eventually). So, on with the definitions ...

Definition 5.21

Suppose f = (<u,Q,s>, P), we start with a few auxilliaries

corresponding to the three categories mentioned above:

in-send(f,R,S) = [t: ∃ e. <t,e> ε mset(Q(R,S,send))]

in-pending(f,R,S) = [t': ∃ t. t ε akning(f,R,S), Q(t)=t']

in-sent(f,R,S) = [t: <t> ε mset(Q(R,S,sent))]

Now just one little definition to help keep things 'clean' (i.e. functional) later on in the proof:

clean(f) iff ∀ t. (Q(t) = 0 iff t ∉ receivers(f))

This means that in a clean configuration Q(t)=0 only when the process identified by t is about to acknowledge a communication.

Thus we can now define

cons(f) iff

1. f = (<u,Q,s>, P), u ε IMAGE

2. ∀ R, S.

waiting(f,R,S) = in-send(f,R,S)
++
in-pending(f,R,S)
++
in-sent(f,R,S)

3. clean(f)

□

At this point we can justify our use of multisets because it is simple to show there can be no repetitions in waiting(f,R,S) so the three multisets on the right-hand side of the equation in 2 are disjoint sets (i.e. no repetitions) if 2 is true. This is expressed in the following Proposition,

Proposition 5.5

Suppose f = (<u,Q,s>, P), u ε IMAGE, then

1. ∀ R, S. norep(waiting(f,R,S))

2. If cons(f) then ∀ R, S.

norep(in-send(f,R,S)),

norep(in-pending(f,R,S)),

norep(in-sent(f,R,S))

and these sets are pairwise disjoint.

□

Two relevant points are noted in the following Proposition,

Proposition 5.6

1. ∀ e. cons([[e]])

2. ∀ e. ∀ f ∈ [[e]]. cons(f)

□

For each consistent experiment f we can now define 'inv(f)' as a source experiment e such that f becomes a member of [[e]] when all communications received but not fully acknowledged are run to completion by both participants. No new actions are started. We will explain a little more fully in the following definition,

Definition 5.22

Suppose cons(f), f = (<u,Q,s>, P), then

inv(f) = (<p,s>, P)

if $u = inter_1 \,||\ldots||\, inter_n$

$p = R_1::c_1 \,||\ldots||\, R_n::c_n$

$match(inter_i,Q) = R_i::c_i$,i = 1,...,n

where

$match([\![R::c]\!]_{(R,t)}, Q) = R::c$

$match(R::wait((R,S,sent),<t>);[\![c]\!]_{(R,t)}, Q) =$

$$\begin{cases} R::c & \text{if } t \in \text{in-pending}(f,R,S)\text{++in-sent}(f,R,S) \\ R::S!e;c & \text{if } \langle t,e\rangle \in \text{mset}(Q(R,S,send)) \end{cases}$$

Note that if only the value of e was stored then match would be very hard to define. That's why the expression e was stored rather than the value.

match(S::akn((R,S,sent),<t>);$[\![c]\!]_{(S,t)}$, Q) = S::c

The reasoning is as follows,

1. inv(f) has the same state as f, any communications which have been received have already had their associated value bound in the state.

2. The progress of any particular labelled command $inter_i$ (or rather the progress of the matching source command) can be found by reference to the communication state. (It should be noted at this point that it is necessary to prove match is indeed a function, this result follows from the consistency of f and will not be pursued further here). Match is defined by case analysis of the definition of IMAGE:

The match of a translated command is of course the command itself.

The match of a sender waiting for acknowledgement depends on whether or not its offer has been received, if so its wait is ended, if not then the expression from the queue is used to reconstruct the sending command whose action placed it on the queue and the communication is deemed not to have taken place.

The match of a receiver about to acknowledge assumes the receive action to be finished

□

The following Proposition establishes some important points about inv,

Proposition 5.7

1. inv(f) is well-defined if cons(f).

2. ∀ e. inv(⟦e⟧) = e

3. ∀ e. ∀ f ∈ ⟦e⟧. inv(f) = e

□

The heart of the proof of conditions c1 to c4 lies in the following Proposition and two Lemmas. Proposition 5.8 will establish condition c2 while Lemma 5.1 and Lemma 5.2 determine the forward and reverse (respectively) implications of condition c1. Conditions c3 and c4 are readily established directly from the appropriate definitions.

Proposition 5.8

If cons(f), f = (⟨u,Q,s⟩, P), f -≠>, then

1. ∀ R, S. waiting(f,R,S) = in-send(f,R,S)

2. ∃ e. f ∈ ⟦e⟧

Proof

1. Suppose there exist R and S such that the consequent is not true. Then, since cons(f), we must have

   in-pending(f,R,S)++in-sent(f,R,S) ≠ ϕ

<u>case</u> in-pending(f,R,S) ≠ ϕ: then

   ∃ t. t ∈ akning(f,R,S)

hence, by the operational semantics,

   ∃ f'. run({t},f,f')

i.e. $f \rightarrow \Rightarrow f'$ (by Proposition 5.3 (2)).

<u>case</u> in-pending$(f,R,S) = \phi$, in-sent$(f,R,S) = \phi$: but we must have

in-sent$(f,R,S) \subseteq$ waiting(f,R,S)

hence

$\exists\, q, t.\ Q(R,S,sent) = \langle t \rangle.q,\ t \in$ waiting(f,R,S)

thus, by the operational semantics,

$\exists\, f'.$ run$(\{t\},f,f')$

i.e. $f \rightarrow \Rightarrow f'$ (by Proposition 5.3 (2)).

2. First we need to establish a few points about f; suppose

$u = inter_1 \,||...||\, inter_n$

then

1. $\forall$ R, S. waiting(f,R,S) = in-send(f,R,S),

akning(f,R,S) = in-sent(f,R,S)

This was established in part 1, except to note that

in-pending$(f,R,S) = \phi$ implies akning$(f,R,S) = \phi$

2. assigners$(f) = \phi$, else by the operational semantics $f \rightarrow$.

Note if eval were not a total function then this might not hold since eval$(e,s) = v$ might not be provable, thus robbing the assignment semantic rule (section 1.3) of its antecedent. However, the result would still hold.

3. $\forall\ t \in$ Nat. $Q(t) = 0$, this follows immediately since clean(f).

Thus, every $inter_i$ is in one of the forms

a) R:: wait(_); $[\![c]\!]_{(R,t)}$

b) $[\![$S::R?x; c$]\!]_{(S,t)}$

c) done

Now, take e = inv(f)

□

In the following arguments we will need to single out from a program (in either Source or Target) the particular labelled command(s) involved in a transition, of which there are at most two. We do this by defining a set of injective functions:

Definition 5.23

A (binary)(Source) program context

pc: Prog x Prog -> Prog

is a (partial) function defined for some

$n \geq 2$,

$i,j \in \{1,\ldots,n\}$, $i \neq j$,

$\{R_k::c_k : k \in \{1,\ldots,n\}-\{i,j\}\}$

by $pc(p_1,p_2) = R_1::c_1 \,||\ldots||\, R_n::c_n = p$

if $p_1 = R_i::c_i$,

$p_2 = R_j::c_j$,

$\vdash p$

Thus a context is essentially a program with two holes for (compatible) labelled commands. A similar definition also serves for unary (Source) program contexts, also ranged over by pc.

For binary Target contexts, ranged over by uc, we must take into account the identifying token of an argument labelled command;

uc: $INTER_t$ x $INTER_u$ -> IMAGE

is a partial function defined, for some

$n \geq 2$,

$t,u \in \{1,\ldots,n\}$, $t \neq u$,

$\{inter_k : k \in \{1,\ldots,n\}-\{t,u\}\}$

by $uc(inter_t, inter_u) = r$

if $r = inter_1 \ ||...|| \ inter_n$,

$\vdash r$

Again, unary contexts are defined similarly and also ranged over by uc.

□

Lemma 5.1

$\forall$ e. (e -> e' implies $[\![e]\!]$ -> => $[\![e']\!]$)

Proof

The proof proceeds by cases on the derivation of e -> e'; we will just demonstrate the difficult case wherein

$e = (\langle pc[R::S!e;c_1, S::R?x;c_2], s\rangle, P)$,

$e' = (\langle pc[R::c_1, S::c_2], s[v/x]\rangle, P)$

where eval(e,s) = v. Then $\exists\ t_1, t_2$.

$[\![e]\!] = (\langle uc[\ [\![R::S!e;c_1]\!]_{(R,t1)}, [\![S::R?x;c_2]\!]_{(S,t2)}\ ],$

initQ,

s>, P)

where uc: $INTER_{t1}$ x $INTER_{t2}$ -> IMAGE

is the binary program context such that $\forall\ p_1, p_2$.

$$uc[\ [\![p_1]\!]_{(R,t1)}, [\![p_2]\!]_{(S,t2)}\ ] = [\![\ pc[p_1,p_2]\ ]\!]$$

whenever both sides are defined. Now, we proceed to 'run' the translation by asserting certain instances of the relation 'run'

Step 1

$run(\{t_1\}, [\![e]\!], f_1)$

where

f_1 = (<uc[R::wait((R,S,sent),<t_1>); [[c]]$_{(R,t1)}$,

[[S::R?x; c_2]]$_{(S,t2)}$],

initQ[<t_1,e>/(R,S,send)],

s>, P)

Step 2

run({t_2}, f_1, f_2)

where

f_2 = (<uc[_,

S::akn((R,S,sent),<t_2>); [[c_2]]$_{(S,t2)}$],

initQ[t_1/t_2],

s[v/x]>, P)

and eval(e,s) = v.

Step 3

run({t_2}, f_2, f_3)

where

f_3 = (<uc[_,

S::[[c_2]]$_{(S,t2)}$],

initQ[<t_1>/(R,S,sent)],

s[v/x]>, P)

Step 4

run({t_1}, f_3, f_4)

where

f_4 - (<uc[R::[[c_1]]$_{(R,t1)}$
S::[[c_2]]$_{(S,t2)}$],
initQ,
s[v/x]>, P)

By the construction of uc,

f_4 = [[e']].

hence

[[e]] -> f_1 -> f_2 -> f_3 -> f_4 = [[e']]

□

Corollary 5.1

∀ e. if e => e' then [[e]] => [[e']]

□

Lemma 5.2

∀ f. if cons(f), f-> f' then

1. cons(f')

2. inv(f) => inv(f')

Proof

We proceed by cases on the derivation of f -> f' (i.e. by cases on IMAGE). In proving 1. we will concentrate on the second part of cons, to the detriment of the first and third, mainly by demonstrating how changes to the configuration 'balance out'.

The proof of 2. will involve proving either inv(f) = inv(f') (cases 2,4 and 5) or inv(f) -> inv(f') (cases 1 and 3) wherein 'things actually get done' by changing the state and, in case 3, making a firm committment to a particular communication.

<u>case</u> 1:

$f = (\langle u,Q,s\rangle, P)$,

$f' = (\langle u',Q,s[v/x]\rangle, P)$,

$u = uc[\ [\![R::x:=e;c]\!]_{(R,t)}\ ]$

$= inter_1 \,||...||\, inter_n$

$u' = uc[\ [\![R::c]\!]_{(R,t)}\ ]$

$= inter'_1 \,||...||\, inter'_n$

$eval(e,s) = v$

1. Clearly $cons(f')$ since none of the related entities changed.

2. Firstly,

$i \neq t$ implies $match(inter_i,Q) = match(inter'_i,Q)$

and thus since

$match([\![R::x:=e;c]\!]_{(R,t)}, Q) = R::x:=e;c$,

$match([\![R::c]\!]_{(R,t)}, Q) = R::c$

we have by Definition 5.22 and the semantics of Source

$inv(f) \Rightarrow inv(f')$

<u>case</u> 2:

$f = (\langle u,Q,s\rangle, P)$

$f' = (\langle u',Q',s\rangle, P)$

$u = uc[\ [\![R::Sle;c]\!]_{(R,t)}\ ]$

$= inter_1 \,||...||\, inter_n$

$u' = uc[\ R::wait((R,S,sent),\langle t\rangle);[\![c]\!]_{(R,t)}\ ]$

$= inter'_1 \,||...||\, inter'_n$

$Q' = Q[\langle t,e\rangle.Q(R,S,send)/(R,S,send)]$

1. The only changes interesting to cons are:

$$waiting(f',T,U) = \begin{cases} waiting(f,R,S)++[t] & \text{if } (T,U) = (R,S) \\ waiting(f,T,U) & \text{otherwise} \end{cases}$$

$$sending(f',T,U) = \begin{cases} sending(f,R,S)++[t] & \text{if } (T,U) = (R,S) \\ sending(f,T,U) & \text{otherwise} \end{cases}$$

So the invariant is preserved because both waiting and sending, with respect to (R,S), is augmented by t.

2. Firstly, from the above identities and the definition of match we get

$i \neq t$ implies $match(inter_i,Q) = match(inter'_i,Q')$

and since

$\langle t,e\rangle \in in\text{-}send(f',Q)$

we get

$match(R::wait((R,S,sent),\langle t\rangle);[\![c]\!]_{(R,t)},Q') = R::S!e;c$

Thus

$inv(f) = inv(f')$.

case 3:

$f = (\langle u,Q[\langle t,e\rangle.q/(R,S,send)],s\rangle, P)$

$f' = (\langle u',Q[q/(R,S,send), t/t'], s[v/x]\rangle, P)$

$u = uc[\ [\![S::R?x;c]\!]_{(R,t')}\ ]$

$= inter_1 \,||\ldots||\, inter_n$

$u' = uc[\ S::akn((R,S,sent),\langle t'\rangle);[\![c]\!]_{(R,t')}\ ]$

$= inter'_1 \,||\ldots||\, inter'_n$

1. The relevant changes are:

$$in\text{-}send(f',T,U) = \begin{cases} in\text{-}send(f,R,S)--[t] & \text{if } (T,U) = (R,S) \\ in\text{-}send(f,T,U) & \text{otherwise} \end{cases}$$

$$in\text{-}pending(f',T,U) = \begin{cases} in\text{-}pending(f,R,S)++[t] & \text{if } (T,U) = (R,S) \\ in\text{-}pending(f,T,U) & \text{otherwise} \end{cases}$$

It is easily deduced that

clean(f')

since

clean(f), t ε receivers(f')

2. We need to show

a) ∀ i ∉ {t,t'}. match($inter_i$,Q) = match($inter'_i$,Q')

b) match($inter_t$,Q) = R::S!e;c'

c) match($inter_{t'}$,Q) = S::R?x;c

d) match($inter'_t$,Q') = R::c'

e) match($inter'_{t'}$,Q') = S::c

so here goes,

a) First, by the operational semantics,

∀ i ∉ {t,t'}. $inter_i$ = $inter'_i$

We proceed by cases on $inter_i$,

<u>case</u> $inter_i$ = $[\![T::c]\!]_{(t'',i)}$:

Trivial since the result is independent of Q, ie

match($inter_i$,Q) = match($inter'_i$,Q') = T::c

<u>case</u> $inter_i$ = T::wait((T,U,sent),<i>);$[\![c]\!]_{(t'',i)}$:

Then from the description of changes in 1. and i ≠ t we must have

in-sent(f,T,U) = in-sent(f',T,U),

i ε in-pending(f,T,U) iff i ε in-pending(f',T,U) ,

in-send(f,T,U) = in-send(f',T,U).

Hence result by the definition of match in this case.

<u>case</u> $inter_i$ = U::akn((T,U,sent),<i>);$[\![c]\!]_{(t'',i)}$:

Trivial, again, since the result is independent of Q.

b) Well,

t ε in-send(f,R,S)

so by cons

$t \in waiting(f,R,S)$

hence

$\exists c'.\ inter_t = R::wait((R,S,sent),\langle t\rangle); [\![c']\!]_{(R,t)}$

thus

$\exists e.\ match(inter_t,Q) = R::S!e;c'$

c) Immediate from the definition of match.

d) Reasoning as in b) we note that now

$t \in in\text{-}pending(f',R,S)$

e) Immediate from the definition of match.

Now clearly

$inv(f) \rightarrow inv(f')$

by the communication from R to S.

<u>case</u> 4:

f = $(\langle u,Q[q/(R,S,sent),\ t/t'],s\rangle,\ P)$

f' = $(\langle u',Q[q.\langle t\rangle/(R,S,sent),\ 0/t'],s\rangle,\ P)$

u = $uc[\ S::akn((R,S,sent),\langle t'\rangle); [\![c]\!]_{(S,t')}\ ]$

= $inter_1\ ||...||\ inter_n$

u' = $uc[\ S::[\![c]\!]_{(S,t)}\ ]$

= $inter_1'\ ||...||inter_n'$

1. As usual we present the relevant changes

$$in\text{-}sent(f',T,U) = \begin{cases} in\text{-}sent(f,R,S)++[t] & \text{if } (T,U) = (R,S) \\ in\text{-}sent(f,T,U) & \text{otherwise} \end{cases}$$

$$in\text{-}pending(f',T,U) = \begin{cases} in\text{-}pending(f,R,S)--[t] & \text{if } (T,U) = (R,S) \\ in\text{-}pending(f,T,U) & \text{otherwise} \end{cases}$$

and of course clean(f') holds.

2. We need to prove

$$\forall\ i \in \{1,\ldots,n\}.\ match(inter_i,Q) = match(inter'_i,Q')$$

If i=t' the result is immediate by the definition of match, and if $inter_i$ is not a waiting process then the result is similarly obtained. So the only case left is

$$inter_i = T::wait((T,U,sent),<t">);[\![c"]\!]_{(T,t")},$$

$$t" = t' \quad \text{(but it's possible } t" = t\text{)}$$

However, from the discussion in 1. we have

$$in\text{-}pending(f,T,U)++in\text{-}sent(f,T,U) =$$

$$in\text{-}pending(f',T,U)++in\text{-}sent(f',T,U),$$

$$in\text{-}send(f,T,U) = in\text{-}send(f',T,U)$$

thus, by the definition of match,

$$match(inter_i,Q) = match(inter'_i,Q')$$

since

$$inter'_i = inter_i$$

Hence

$$inv(f) = inv(f')$$

<u>case</u> 5:

f= (<u,Q[<t>.q/(R,S,sent)],s>, P)

f'= (<u',Q[q/(R,S,sent)],s>, P)

u= uc[R::wait((R,S,sent),<t>);[\![c]\!]$_{(R,t)}$]

u'= uc[R::[\![c]\!]$_{(R,t)}$]

This case presents no further difficulties, suffice it to say

inv(f) = inv(f').

□

Corollary 5.2

$\forall$ f. if cons(f), f => f', then

1. cons(f')
2. inv(f) => inv(f')

□

We now turn to the proof of conditions c1 to c4 given our accumulated armoury of Propositions, Lemmas and Corollaries,

Theorem 5.2

$[\![.]\!]_2$: SEXP -> TEXP is a complete implementation.

Proof

By Proposition 5.1, SEXP and TEXP are computationally finite and success terminating so we prove conditions c1 to c4 in the statement of Theorem 5.1

c1. Suppose e => e', then by Corollary 5.1

$[\![e]\!] \Rightarrow [\![e']\!]$

and by Proposition 5.4

$\exists$ f. $[\![e']\!] \Rightarrow f \in [e')$

Now suppose $\exists$ f. $[\![e]\!] \Rightarrow f \in [\![e)$, then by Proposition 5.6 and Corollary 5.2

$inv([\![e]\!]) \Rightarrow inv(f)$

but by Proposition 5.7

$inv([\![e]\!]) = e$, $inv(f) = e'$

thus

e => e'

c2. By Proposition 5.6, Corollary 5.2

cons(f)

and thus from Proposition 5.8

$\exists$ e'. f ε [[e']

c3. Follows by simple case analysis on the proof of the derivation.

c4. Immediate from the definitions.

□

5.2.4. Alternative results

In this section we have shown that any translation between computationally finite success terminating experiment systems which satisfies conditions c1 to c4 is a complete implementation (proved in Theorem 5.1). Though an interesting result (we hope!) there is no reason, beyond its use in the current example, why it should stand out from many similar results as 'the one to be proved'. We would like to suggest that what is important here is the use of an 'abstract layer', underneath the testing definition of translation correctness, which emphasises properties of experiment systems and their computations. This allows a flexible 'mix and match' approach to verifying translations. For instance we can prove a translation is an implementation (only) with a similar set of conditions to c1 to c4, by employing the following Definition and Theorem;

Definition 5.24

Suppose <E,->,success> is an experiment system, e ε E, then define div(e) iff e has an infinite computation, i.e.

div(e) iff $\exists$ $\langle e_n \rangle_{Nat}$ ε comp(e). e = e_0 -> e_1 -> ...

□

Theorem 5.3

Suppose

1. $S = \langle E_S, \rightarrow_S, success_S \rangle$,e ranges over E_S,

   $T = \langle E_T, \rightarrow_T, success_T \rangle$,f ranges over E_T

   are success terminating experiment systems,(subscripts will be dropped from now on where unnecessary).

2. $[\![.]\!]: E_S \rightarrow E_T$, $[\![.)\!]: E_S \rightarrow P(E_T)$

then

C1. $\forall$ e, e'. ($\nexists$ f ε $[\![$e'$)\!]$. $[\![$e$]\!]$ $\Rightarrow$ f) implies e $\Rightarrow$ e'

C2. $\forall$ e, f. $[\![$e$]\!]$ $\Rightarrow$ f $\not\rightarrow$ implies $\exists$ e'. f ε $[\![$e'$)\!]$

C3. $\forall$ e. $\forall$ f ε $[\![$e$)\!]$. e $\rightarrow$ iff f $\rightarrow$

C4. $\forall$ e. $\forall$ f ε $[\![$e$)\!]$. success(e) iff success(f)

C5. $\forall$ e. div($[\![$e$]\!]$) implies div(e)

implies

$\forall$ e. $[\![$e$]\!]$ <u>implements</u> e.

□

Note that the demand on computational finiteness has been omitted and we have dropped the forward implication in c1 and added C5 to handle infinite computations (Actually we only need the forward implication in C3). The proof proceeds as in Theorem 5.1,

Proof Let e ε E

(e <u>must</u> implies $[\![$e$]\!]$ <u>must</u>): Suppose e <u>must</u>; take any computation of $[\![$e$]\!]$, then it must be finite since by C5

div($[\![$e$]\!]$) iff div(e)

which contradicts e <u>must</u> since E_S is success terminating. Thus we can proceed, as before, on the finite case,

$[\![$e$]\!]$ $\Rightarrow$ f $\not\rightarrow$

then by C2

$\exists$ e'. f ε $[\![$e'$)\!]$

so by C1

e => e'

and by C3

e' -/>

Thus since E_S is success terminating and e must (by hypothesis) we have

success(e')

and hence by C4

success(f)

([[e]] may implies e may): Then since E_T is success terminating we have

∃ f. [[e]] => f -/>, success(f)

Now, by C2

∃ e'. f ε [[e)

so by C1 and C4

e => e'

□

Of course, another similar result would be that for computationally finite success sustaining systems conditions c1 to c4 lacking the forward implication of condition c1 are sufficient to guarantee an implementation.

Condition C5 is the only one we haven't already proved for our example translation, simply because there are no infinite computations in SEXP or TEXP. If we were to add some kind of iterative or recursive construct to the Source language then we might like to establish this condition for our new translation. One way of doing this is through the following Proposition (given

extended definitions of cons, inv etc)

Proposition 5.9

Let f,f' range over config(TEXP) and suppose for all f such that cons(f).

$$\exists\, k \geq 1.\ f \rightarrow_k f' \text{ implies } inv(f) \rightarrow \Rightarrow inv(f')$$

then

$$\forall\, e.\ div([\![e]\!]) \text{ implies } div(e)$$

Proof Simply noting cons([[e]]) we see that there must be an infinite computation

$$[\![e]\!] \rightarrow_k f_1 \rightarrow_k f_2 \rightarrow_k \cdots$$

so there must be an infinite computation

$$e \rightarrow \Rightarrow inv(f_1) \rightarrow \Rightarrow inv(f_2) \rightarrow \Rightarrow \cdots$$

□

In this Proposition the number k sets an upper bound on how many 'housekeeping' steps the translation can make before doing some of the real work performed by the original. In our proof the real work was done when the state was changed by assigning some value to a variable during either an assignment statement or a receipt (rec(qvd,<t',x>)) statement. Whenever some experiment f performed one of these statements in a derivation f -> f' we showed inv(f) -> inv(f'), whereas for the other statements inv(f) = inv(f') so only (necessary!) 'housekeeping' work has been performed. For the example translation it is easily seen that if

$$f = (\langle r_1 \,||\ldots||\, r_n, s\rangle,\ P)$$

then taking k=3n establishes the antecedent of Proposition 5.9.

A more general version of this approach would be to replace C5 by the following condition:

C5.1 $\forall$ e. $\exists$ k $\geq$ 1. $\forall$ f.

($[\![e]\!] \rightarrow_k f$ implies $\exists$ e'. e $\rightarrow\Rightarrow$ e', $[\![e']\!] \Rightarrow$ f)

i.e. if a translated configuration takes k steps to reach f then it must have simulated at least one source action. Then we endeavour to prove C5 by:

Proof of C5:

Suppose div($[\![e]\!]$), then

$\exists$ f. $[\![e]\!] \rightarrow_k$ f, div(f)

for that k provided by C5.1. Hence

$\exists$ e_1. e $\rightarrow\Rightarrow$ e_1, $[\![e_1]\!] \Rightarrow$ f

so div($[\![e_1]\!]$)

and the construction can be repeated from e_1 to produce e_n for any n, giving an infinite computation

e $\rightarrow\Rightarrow$ e_1 $\rightarrow\Rightarrow$ e_2 $\rightarrow\Rightarrow$ ...

□

To establish C5.1 for our (extended) translation would probably be quite simple using (an extended) Lemma 5.1. However, finding a general condition to establish C5.2 stated in terms akin to C5.1 may be problematic since Lemma 5.1 says something about the configurations which a translation may take up <u>during</u> a computation rather than just the end points characterised by $[\![.]\!]$ and $[\![.)$. For our example (and hopefully the extended version) it is the case that configurations which are also translations of source configurations can arise almost as required in a computation but this is not to be expected in general.

A proof of complete implementation for the infinite case would require adding the converse (C1.1) of C1 and the converse (C5.2)

of C5 to the conditions C1,...,C5:

C1.1 ∀ e, e'. e => e' implies ⫞ f ∈ [[e']. [[e]] => f

C5.2 ∀ e. div(e) implies div([[e]])

then we can prove ∀ e. [[e]] must implies e must as follows:

Proof of must:

Suppose [[e]] must; take any complete computation (finite or infinite) of e

case e => e' -⊬>: then by C1.1

⫞ f ∈ [[e']. [[e]] => f

and by C2

f -⊬>

Thus since [[e]] must

success(f)

and then by C4

success(e')

case div(e): then by C5.2

div([[e]])

which is not possible since [[e]] must in a success terminating system.

□

To prove ∀ e. e may implies [[e]] may :

Proof of may:

Suppose e may; then we have a finite computation

e => e', success(e')

hence by C1.1

⫞ f ∈ [[e']. [[e]] => f

and by C4

success(f)

□

5.2.5. Extending the translation

Having translated a very small core of CSP to a specialllly devised language the question of possible extensions to the translation becomes intertwined with the question of possible extensions to the Target language. Thus we should concentrate mainly on the translation method in discussing extensions of the translation and the proof of correctness. Perhaps the most important missing command forms are the conditional (if gc fi), repetitive (do gc od) and (nested) parallel (c1||c2) forms.

if gc fi: It would seem the main problem here lies in translating guarded commands like

$b_1 \Rightarrow R_1!e_1 \;\square\; b_2 \Rightarrow R_2!e_2 \;\square\; b_3 \Rightarrow R_3!e_3$

wherein the first action of the translation of $R_i!e_i$ will be to put an offer in a send queue. For the above guarded command three offers should be put in queues, but at most one should be accepted. Thus there are two related subproblems

1. Making a communication offer should not select a guarded command for execution.

2. Only one communication offer should be accepted.

do gc od: Aside from the problems of translating the guarded command there is the difficulty of the infinite computations introduced into the experiment systems by this iterative command. This point was discussed earlier, where it was shown how the proof technique might be extended to handle such cases.

c1||c2 : It seems that this construct would pose no difficulties for this translation method in general, however it really all depends on the chosen target language.

6. Weavesort; an example implementation

This chapter has three main aims :

1. To introduce a CCS operator ⊕ as an aid to specification, allowing freedom of choice to the implementer and ease of description to the specifier.

2. To provide an example wherein the relationship of complete implementation, or of having a bisimulation, does not hold in a natural way between the described process and its specification.

3. To show how object implementation may be proved using a technique similar to that employed in the bisimulation theory.

6.1. Extending EXP(CCS) to NEXP(CCS)

Since our aim in this chapter is to establish the relation of object implementation (see Definition 4.11 chapter 4 section 2.5) between a sorting machine and its specification we need to define an experiment system in which the former implements the latter. The system we choose is very similar to EXP(CCS) (see chapter 4 section 1) so experiments are CCS terms, the transition relation is $-\stackrel{1}{\rightarrow}$, and success is the ability to perform the distinguished action ω. However, we need to extend EXP(CCS) (to NEXP(CCS)) in two ways:

1. Adding indeterminacy; we will need to express in a single CCS term a number of possible processes, so that whenever a term is encountered an arbitrary choice can be made between the

possibilities. This will greatly aid both the description and specification of the sorting machine.

2. Extending the set of experiments: an experiment in EXP(CCS) is of the form 'test|object', so given an experiment it is always clear which part is the test and which part is the object. Part of our proof technique will involve blurring the distinction between test and object, and to this end we allow an experiment to be (loosely) a term one of whose subterms is the object. The remainder is the test.

Unfortunately we will find it necessary to restrict Val, the set of communicable values, to be a finite set. This requirement comes up at two distinct places and is pointed out as it arises.

6.1.1. Adding indeterminacy

To aid in the specification and implementation of behaviours in CCS, and thereby NEXP(CCS), we introduce the notion of indeterminacy via a "don't care" or "don't know" operator '⊕' which we add to the CCS of chapter 1. In the interests of brevity we will not completely describe the new CCS, but rather concentrate on the changes.

We expand the possible CCS terms by adding a new operator ⊕ to those already present; if p,q are terms then p⊕q is a term. This term p⊕q represents either the process represented by the term p or the process represented by the term q. The difference between p+q and p⊕q is rather subtle. However, the former represents a process which is altogether different from both the processes

represented by p and q, whereas the latter represents precisely one or the other; it is just that we don't know (or care) which.

The rules for inferring transition relationships are very similar to those given in chapter 1 but the new operator ⊕ must be accomodated. For example we would like the relations $-\underline{a}\rightarrow$ to be such that

$$A!.(B! + C!) -\underline{A!}\rightarrow p \quad \text{iff} \quad p = B! + C!$$

but

$$A!.(B! \oplus C!) -\underline{A!}\rightarrow p \quad \text{iff} \quad p = B! \text{ or } p = C!$$

Thus whenever a term p⊕q comes to be executed we want to replace it with either p or q (we don't care which) and execute that. We achieve this by introducing the notion of a 'deterministic state' of a term.

Definition 6.1

Suppose we are employing a definition set D such that

$$P(x_1,\ldots,x_n) \Leftarrow q$$

For any term p let ds(p) be the set of terms defined by induction on p as follows:

$$ds(Nil) = Nil$$

$$ds(a.p) = \{a.p\}$$

$$ds(p_1+p_2) = \{p_1'+p_2' : p_1' \in ds(p_1),\ p_2' \in ds(p_2)\}$$

$$ds(p_1 \oplus p_2) = ds(p_1) + ds(p_2)$$

$$ds(p_1|p_2) = \{p_1'|p_2' : p_1' \in ds(p_1),\ p_2' \in ds(p_2)\}$$

$$ds(p[E]) = \{p'[E] : p' \in ds(p)\}$$

$$ds(\text{if } b \text{ then } p \text{ else } q) = \begin{cases} ds(p) & \text{if } eval(b) = tt \\ ds(q) & \text{if } eval(b) = ff \end{cases}$$

$$ds(P(e_1,\ldots,e_n)) = ds(q[v_1/x_1,\ldots,v_n/x_n])$$

where $eval(e_i) = v_i$, $i = 1,\ldots,n$.

so for example

$$ds(B! + C!) = \{B! + C!\}$$

whereas

$$ds(B! \oplus C!) = \{B!, C!\}.$$

Note however that

$$ds(A!.(B! \oplus C!)) = \{A!.(B! \oplus C!)\}$$

so ds only looks at the topmost level of a process. Note also that ds(p) is finite and non-empty for any p.

The relations $-\underline{a}\rightarrow$ are now defined so that whenever $p -\underline{a}\rightarrow q$, q is a deterministic state, that is $ds(q)=\{q\}$ (We may say q is determinate, or a determinate term). So a computation evolves by a process changing, under the stimulus of actions, from deterministic state to deterministic state. However, a computation may begin from a nondeterministic state and it is only for this state that the derivation rule, given below, for $\oplus$ is required.

Definition 6.2

Let $-\underline{a}\rightarrow$ be the least relation contained in Terms x Terms such that the following rules hold:

1. **Inaction**

   Nil has no transitions.

2. **Action** (different from chapter 1)

   2.1 $1.p -\underline{1}\rightarrow p'$ if $p' \in ds(p)$

   2.2
   $$\frac{eval(e)=v,\ p' \in ds(p)}{A!e.p -\underline{A!v}\rightarrow p'}$$

   2.3 $A?x.p -\underline{A?v}\rightarrow p'[v/x]$ if $p' \in ds(p)$.

3. **Choice** (as chapter 1)

$$3.1\quad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad 3.2\quad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'}$$

4. **Indeterminacy**

$$4.1\quad \frac{p \xrightarrow{a} p'}{p \oplus q \xrightarrow{a} p'} \qquad 4.2\quad \frac{q \xrightarrow{a} q'}{p \oplus q \xrightarrow{a} q'}$$

This definition may seem a little odd at first since it gives exactly the same derivations to $p \oplus q$ and $p+q$. However, it can be justified on the grounds that it is only used in the first step of a computation. In the testing theory we are interested in the set of all possible computations of an experiment (which in this case is a term) and not (so much) in the particular computation we get in a single trial. Hence, the fact that we have given

(1): $((A! \oplus B!) \mid A?) \backslash A$

only the single computation

$$((A! \oplus B!) \mid A?) \backslash A \xrightarrow{1} (Nil \mid Nil) \backslash A$$

which is essentially identical with the only computation of

(2): $((A! + B!) \mid A?) \backslash A$

is (for our purposes) redressed by the following convention.

Convention The computations in comp(p) of length zero are precisely the determinate terms $p' \varepsilon ds(p)$ such that $p' \not\rightarrow$.

Then (1) has a single computation of length zero, viz.

$(B! \mid A?) \backslash A$

while (2) has none.

□

5. **Parallel** (different from chapter 1)

5.1 $$\frac{p \xrightarrow{a} p',\ q' \in ds(q)}{p|q \xrightarrow{a} p'|q'}$$

5.2 $$\frac{q \xrightarrow{a} q',\ p' \in ds(q)}{p|q \xrightarrow{a} p'|q'}$$

The substitution of determinate terms is necessary so that all derivatives are determinate.

5.3 $$\frac{p \xrightarrow{a} p',\ q \xrightarrow{\bar{a}} q'}{p|q \xrightarrow{1} p'|q'}$$

6. **Renaming** (as chapter 1)

6.1 $$\frac{p \xrightarrow{a} p'}{p[E] \xrightarrow{F[a]} p'[E]}$$

if F(a) is defined, where

$$F(A) = \begin{cases} 1 & \text{if } a = 1 \\ E(A)!v & \text{if } a = A!v,\ E(A) \text{ defined} \\ E(A)?v & \text{if } a = A?v,\ E(A) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

7. **Conditional** (as chapter 1)

7.1 $$\frac{beval(b)=tt,\ p \xrightarrow{a} p'}{\text{if } b \text{ then } p \text{ else } q \xrightarrow{a} p'}$$

7.2 $$\frac{beval(b)=ff,\ q \xrightarrow{a} q'}{\text{if } b \text{ then } p \text{ else } q \xrightarrow{a} q'}$$

Note that by Definition 6.1 these rules will only be used at the beginning of a computation.

8. **Procedure call** (as chapter 1)

Suppose $D(P(x_1,\ldots,x_n)) = p$ is defined, then

$$\frac{p[v_1/x_1,\ldots,v_n/x_n] \xrightarrow{a} p'}{P(e_1,\ldots,e_n) \xrightarrow{a} p'}$$

where $eval(e_i) = v_i$, $i = 1,\ldots,n$

Note that, in common with the rules for $\oplus$, this rule will only be used at the beginning of a computation since the function ds substitutes the body of the call immediately on encounter. It is important to note that in what follows we will not bother to distinguish the body from the call, employing the call as a description of the body. This is permissible because in only one case will the body of the call (to $WS^n(M)$) not be a determinate term and for that case we will distinguish body from call.

□

The following proposition holds;

Proposition 6.1

$p \xrightarrow{a} p'$ iff $\exists\, dp \in ds(p).\ dp \xrightarrow{a} p'$

□

Next we introduce a little notation to enable simple description of complicated terms,

Notation In analogy with + and $\sum$ we introduce

$\bigoplus_{k \in K} q_k$ to denote $q_{k1} \oplus \ldots \oplus q_{kn}$

where $K = \{k_1,\ldots,k_n\}$ is some finite index set for $n \geq 1$. This is permissible since $\oplus$ will behave at all times like an associative commutative operator.

□

The definitions so far are fine if we are dealing with determinate terms but we really want to talk about indeterminate terms during

the course of a computation. From the definition of the transition relations $\xrightarrow{a}$ we have seen such terms don't occur within a computation. However, it will be useful in the proof of the may part of implementation to behave as if they did occur; this will allow us to defer decisions about which of a number of possible processes has been chosen, we will know it is one of the definite states but not which one.

We really want to act as if we had a slightly different transition relation wherein indeterminacy can remain unresolved arbitrarily far past its first encounter, the essential idea being that if $p \xrightarrow{a} r$ and $p \xrightarrow{a} q$ then if $p \xrightarrow{a} p'$ it's possible that $p' = q$ or $p' = r$, which we would like to write as

$$p \xrightarrow{a} q \oplus r$$

which may be read "p may perform the action a, thereby becoming one of q or r". We define this new transition relation between (indeterminate) terms as follows:

Definition 6.3

$$p \xrightarrow{a} q \quad \text{iff} \quad \forall\, dq \in ds(q).\ p \xrightarrow{a} dq$$

Note ds(q) is always non-empty and if q is determinate then the relationship is as before (since $ds(q) = \{q\}$ in that case).

□

Thus for example: $1.(p \oplus q) \xrightarrow{1} p \oplus q$ and $(1.p + 1.q) \xrightarrow{1} p \oplus q$.

At this point we should emphasise that the 'true' transition relation is $\rightarrow$ whereas $\rightarrow$ will be useful for establishing relationships in $\rightarrow$ such as:

Proposition 6.2

$p \xrightarrow{1} \xrightarrow{1}{}^{*} q$ implies $\forall\, dq \in ds(q).\ p \xrightarrow{1}{}^{*} dq$

Proof by induction on the length of the derivation $\xrightarrow{1}{}^{*}$ using Proposition 6.1.

□

6.1.2. Extending the set of tests

So far we have decided that an experiment is a CCS term (possibly indeterminate) and that success is achieved when that term reaches a derivative which offers the distinguished action ω. Since we are dealing in this chapter with the relation <u>implements</u> between objects rather that experiments it is necessary to separate test and object in an experiment. To this end we introduce tests as contexts and objects as terms not containing the action ω, so only the test part may signal success.

A test t, often written t[.] to distinguish it as such, is a CCS term with a single distinguished parameterless procedure call to its argument. Interfacing a test involves substitution of this procedure call by the argument.

Definition 6.4

Suppose we have a distinguished procedure name Arg ε Proc, then a test t is a closed term with a single occurrence of Arg as a subterm. Writing t[p] for t[p/Arg] define the set T of tests, ranged over by t, by

$$t ::= \quad Arg \mid p|t \mid t[E] \mid t_1[t_2]$$

where p ranges over Terms. We will assume each test t is syntactically valid, i.e. $\vdash$ t. Arg is assumed to have no

definition, its occurrence serves merely as a marker for the point of substitution. We may write [.] for Arg when describing tests, thus for example

p | Arg may be written p | [.]

□

The objects to be tested are valid CCS terms not involving ω, so the signal of success must come from the test.

Definition 6.5

The set P of objects is defined by

$$P = \{p: \omega \notin FL(p), \vdash p\}$$

□

The application of a test to an object is of course achieved by substituting the object for Arg, yielding a CCS term (an experiment) with the following property:

Proposition 6.3

If $t \in T$ and $p \in P$ then

$$ds(t[p]) = \{t'[p']: t' \in ds(t), p' \in ds(p)\}$$

□

Now, in analysing and synthesising transitions of experiments it is useful to separate the actions of test and object. To this end we introduce transition relations for contexts.

Definition 6.6

Suppose we add to our definition set D the definitions

$$Test_a \Leftarrow a.1.Did_a$$

$$Did_a \Leftarrow Nil$$

for every a ε AD (see chapter 1 section 2.2), assuming $(Test_a,0)$, $(Did_a,0) \notin Defined(D)$, then for every possible action we have a process $Test_a$ which offers it alone. The derivation rules for tests are then as follows for t ε T:

1. $$\frac{t[Test_a] \xrightarrow{\alpha} t'[1.Did_a]}{t[.] \xrightarrow{\alpha[a]} t'[.]}$$

The consequent should be read as

> If t's argument may perform action a then the whole term may perform action α as t becomes t', a test for whatever the argument has become.

The reason for not defining

$$Test'_a \Leftarrow a.Did_a$$

is that from our definition of ds and the action rules,

$$Test'_a \xrightarrow{a} Nil$$

whereas

$$Test_a \xrightarrow{a} 1.Did_a$$

The second relationship is better for keeping track of the point of substitution, for the subterm $1.Did_a$ will remain at the point of substitution of $Test_a$ but Nil may occur at many points in a term t'[Nil] so we might have

$$t'[Nil] = t''[Nil],\ t' \neq t''$$

in which case we have lost track of what the test was. Proposition 6.4 (1) will affirm the property we require.

Note that the consequent of rule 1 is not intended to describe a new form of action or transition relation for general CCS terms, it is merely a device for describing how a test interacts with its

argument.

In the case $\alpha = 1$, which is the most interesting to us, we write

$$t[.] \xrightarrow{[\underline{a}]} t'[.]$$

2. $$\frac{t[Test_a] \xrightarrow{\underline{\alpha}} t'[Test_a]}{t[.] \xrightarrow{\underline{\alpha}} t'[.]}$$

The consequent may be read

> Regardless of t's argument the whole term performs action α, t becoming t' and the argument remaining unchanged.

□

The definition of test transitions now enables, and is justified by, the following proposition addressing the relationship between test and object in a single transition step of an experiment.

Proposition 6.4

Suppose $t \in T$, $p \in P$ then

1. $t[Test_a] \xrightarrow{\underline{\alpha}} q$ implies

   $\dashv$ unique t'. $(q = t'[Test_a]$ or $q = t'[1.Did_a])$

2. $t[.] \xrightarrow{\underline{a}} t'[.]$ implies

   $\forall\, p \in P.\ \forall\, dp' \in ds(p).\ t[p] \xrightarrow{\underline{a}} t'[dp']$

3. $t[.] \xrightarrow{[\underline{1}]} t'[.]$ implies $t' \in ds(t)$

4. $t[p] \xrightarrow{\underline{1}} q$ iff $\dashv$ t',p'. $q = t'[p']$ and one of the following holds

   1. $t[.] \xrightarrow{\underline{1}} t'[.]$, $p' \in ds(p)$

   2. $\dashv$ a. $t[.] \xrightarrow{[\underline{a}]} t'[.]$, $p \xrightarrow{\underline{a}} p'$

□

From now on we will describe the progression of experiments as

$$t[p] \rightarrow t'[p'] \rightarrow t''[p''] \rightarrow \ldots$$

as justified by Proposition 6.4 (4).

We are now in a position to define NEXP(CCS) and refer the reader to chapter 4 section 1.1 for general motivation.

Definition 6.7

An experiment is the application of a test to an object:

$$exp(NEXP(CCS)) = \{t[p]: t \in T, p \in P\}$$

An experiment is successful if it offers ω:

$$succ(NEXP(CCS)) = \{e: e \in exp(NEXP(CCS)), e \xrightarrow{\omega!}\}$$

An experiment proceeds by the derivation rules of CCS :

$$trans(NEXP(CCS)) = \xrightarrow{1}$$

□

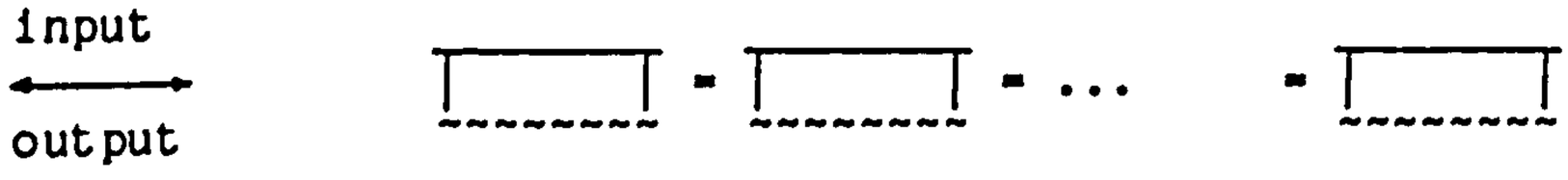
6.2. The weavesort machine

The example we have chosen is taken from the area of VLSI design wherein great emphasis is placed on regularity of structure and a high degree of concurrency. The task to be performed by the machine is that of sorting a multiset of natural numbers into numerical order. Of course the machine will be of finite capacity so there is a limit to the size of the multiset.

6.2.1. An informal description of weavesort

A weavesort machine of size n, capable of sorting a multiset of 2n values, is a linear array of n sorting cells each of which may

hold at most 2 values. The machine is assumed to be initially empty so each cell is empty. Diagrammatically this state is represented as



where each box represents an empty cell, and the connections between cells are written '='. Each cell, except the rightmost, is connected to its right neighbour by a data path across which values can pass in either direction. Data enters and leaves the array through the left cell only, the rightmost cell only communicates with its left neighbour.

The action of this machine is most easily understood in terms of the actions of its constituent cells. The action of a single cell depends on how many values it holds in its two registers (a left register to hold the smaller value and a right register to hold the larger value (if two are values are held)) according with the following rules.

R.0. An empty cell is willing to receive a value from its left and become a cell with one value.

R.1.1.A cell with one value is willing to receive another value from its left and become a cell with two values, the smaller in its left register and the larger in its right.

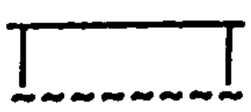
R.1.2.Alternatively, a cell with one value is also willing to deliver its only value to the left and thereby become a cell with no values.

R.2.1.A cell with two values (the smaller in its left register, the larger in its right) is willing to receive a value from its left and then pass the value in its right register to the right. (Note that the cell is momentarily holding three values, a point we shall return to later), thereby becoming a cell with the smaller of its remaining values in its left register and the larger in its right.

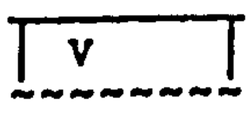
R.2.2.Alternatively, a cell with two values is also willing to deliver its smaller value to the left and then request a value from its right. If the request is granted then it remains a cell with two elements, shuffling its two values to get the smaller in its left register. If the request is denied then it becomes a cell holding its one remaining value.

R.3. If not requested to perform any action it can perform a cell (of any number of values) will do nothing.

An example should make this clear; suppose n = 2 so we have two cells and can sort multisets of up to 4 elements. In this informal presentation we will describe the progress of the machine diagrammatically where



is a cell with no values.



is a cell with the single value v in its left register.

| u v | is a cell with two elements; u (≤ v) in its left register and v in its right.

$d_1 \xrightarrow{v!} d_2$ means v is output (to the left) by the machine (diagram) d_1, thereby becoming d_2.

$d_1 \xrightarrow{v?} d_2$ means v is input (from the left) by the machine (diagram) d_1, thereby becoming d_2.

In describing a computation we will write under each cell of d_1 the rule it applies for that transition.

So, suppose we want to sort the multiset [1,2,3,4]; then all we have to do is input them in any order to the empty machine (of size 2 in our case) and then accept the values it offers back:

	=	
R.0		R.3

$\xrightarrow{2?}$

2	=	
R.1.1		R.3

$\xrightarrow{4?}$

2 4	=	
R.2.1		R.0

$\xrightarrow{1?}$

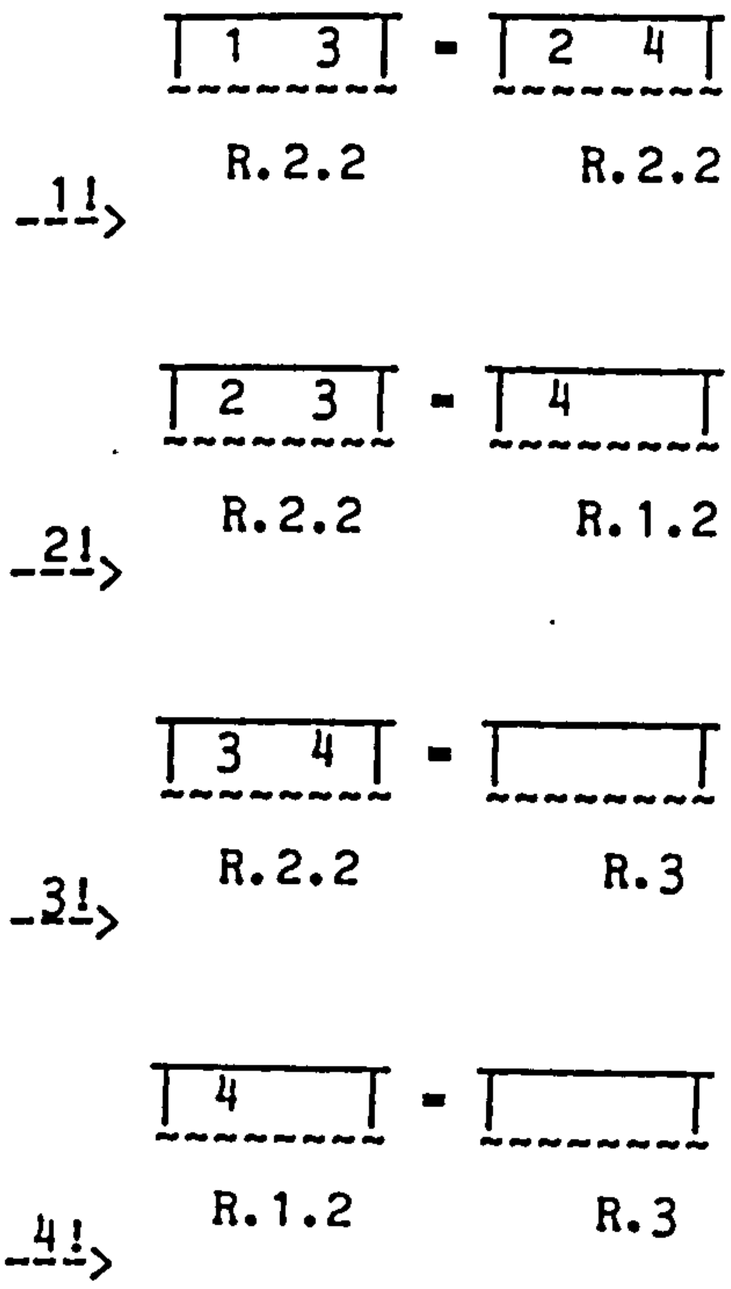
1 2	=	4
R.2.1		R.1.1

$\xrightarrow{3?}$

1 3	=	2 4

Note the machine is now full so we must now start removing

elements. The subject of what happens if we try to keep feeding in more values is deliberately under-defined in this informal description.



6.2.2. The description and specification in CCS

The aim of this subsection is to formalise, in CCS, the informal description of the weavesort machine introduced in the previous subsection. For the purposes of explanation we will find it useful to have a close relationship between the diagrammatic descriptions used earlier and the CCS terms we will define. Each sorting-cell-agent we define can be viewed diagrammatically as

where we may refer to the left as the 'front' and

l,r are value passing channels modelling those data paths to the left and right (respectively) of a cell.

emptyl,emptyr are used for signalling and sensing (respectively) an empty cell.

x,y are bound variables modelling the two registers of a sort cell; x will hold the smaller value if two values are held.

A weavesort machine of size n containing a multiset M of values ($|M| \leq 2n$) is a linear array of n sorting cells over which the elements of M are distributed in a particular fashion. The description of such a machine, to be written $WS^n(M)$ is best understood inductively on its size n and thence by case analysis on its contents M :

BASE n = 0 : The machine is willing to receive values from its left but immediately forgets them. However, it is always willing to admit it is empty (i.e. it contains no values). We describe it in CCS by an agent 'stopper' and diagrammatically by '=|' :

$$\text{stopper} \Leftarrow l?z.\text{stopper} + \text{emptyl}!.\text{stopper}$$

STEP Suppose we have weavesort machines of size n-1 for all multisets of size up to 2(n-1) : we construct weavesort machines of size n for multisets M of size up to 2n by case analysis on M. The (inductive) aim of the construction is always to keep the

smallest value held in the leftmost register of the leftmost cell.

<u>case</u> M = ϕ: We want an 'empty cell' 'connected' to a 'smaller weavesort machine holding no values'. Thus if we describe

'empty cell' by CELLO

'connected' by <=>

'smaller weavesort machine holding no values'

by $WS^{n-1}(\phi)$

then, first in CCS and then diagrammatically, we want

CELLO <=> $WS^{n-1}(\phi)$

We still need to define <=> and CELLO in CCS, beginning with the former which is a notational convention:

Definition 6.8

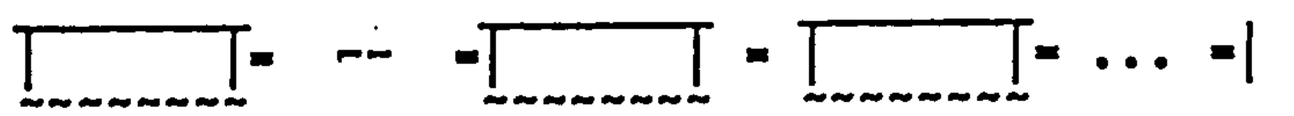
Let p,q be CCS terms, then

p <=> q = (p[m,emptym/r,emptyr] | q[m,emptym/l,emptyl])\{m,emptym}

□

The intended use of <=> is that q is some smaller sorting machine, a linear array of cells, and p is a cell to go on the front; i.e.

p <=> q



So we connect q's left value channel l to p's right value channel r and q's empty-signalling channel emptyl to p's empty-sensing channel emptyr. Since <=> acts like an associative operator the

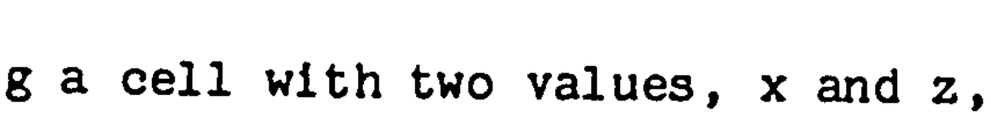
above diagrammatic description of q is permissible.

Now, an empty cell can either receive a value from the left, thereby becoming a cell holding that value, or it can signal to the left that it is empty (since our construction will guarantee that everything to its right is also empty) :

$$\begin{aligned} \text{CELL0} \Leftarrow\ & l?z.\text{CELL1}(z) \\ & + \\ & \text{empty}l!.\text{CELL0} \end{aligned}$$

<u>case</u> M = [m]: we want a cell containing just m, the smallest value in M, connected to a smaller machine holding no values; ie

$$\text{CELL1}(m) \iff WS^{n-1}(\phi)$$



A cell with a single value x can receive another value z, thereby becoming a cell with two values, x and z, the smaller being in the left register. Alternatively, it can output x to the left and become a cell with no values (our construction will guarantee that everything to its right is empty).

$$\begin{aligned} \text{CELL1}(x) \Leftarrow\ & l?z.\text{SWAP}(x,z) \\ & + \\ & l!x.\text{CELL0} \end{aligned}$$

and

$$\text{SWAP}(x,y) \Leftarrow \text{if } x < y \text{ then } \text{CELL2}(x,y) \text{ else } \text{CELL2}(y,x)$$

<u>case</u> M = M'++[m,v], m = min(M): This is the most interesting case. The leftmost cell must hold the smallest value m and some (in fact any) other value v from M. The remainder of M, i.e. M', is distributed recursively over $WS^{n-1}(M')$; thus we want:

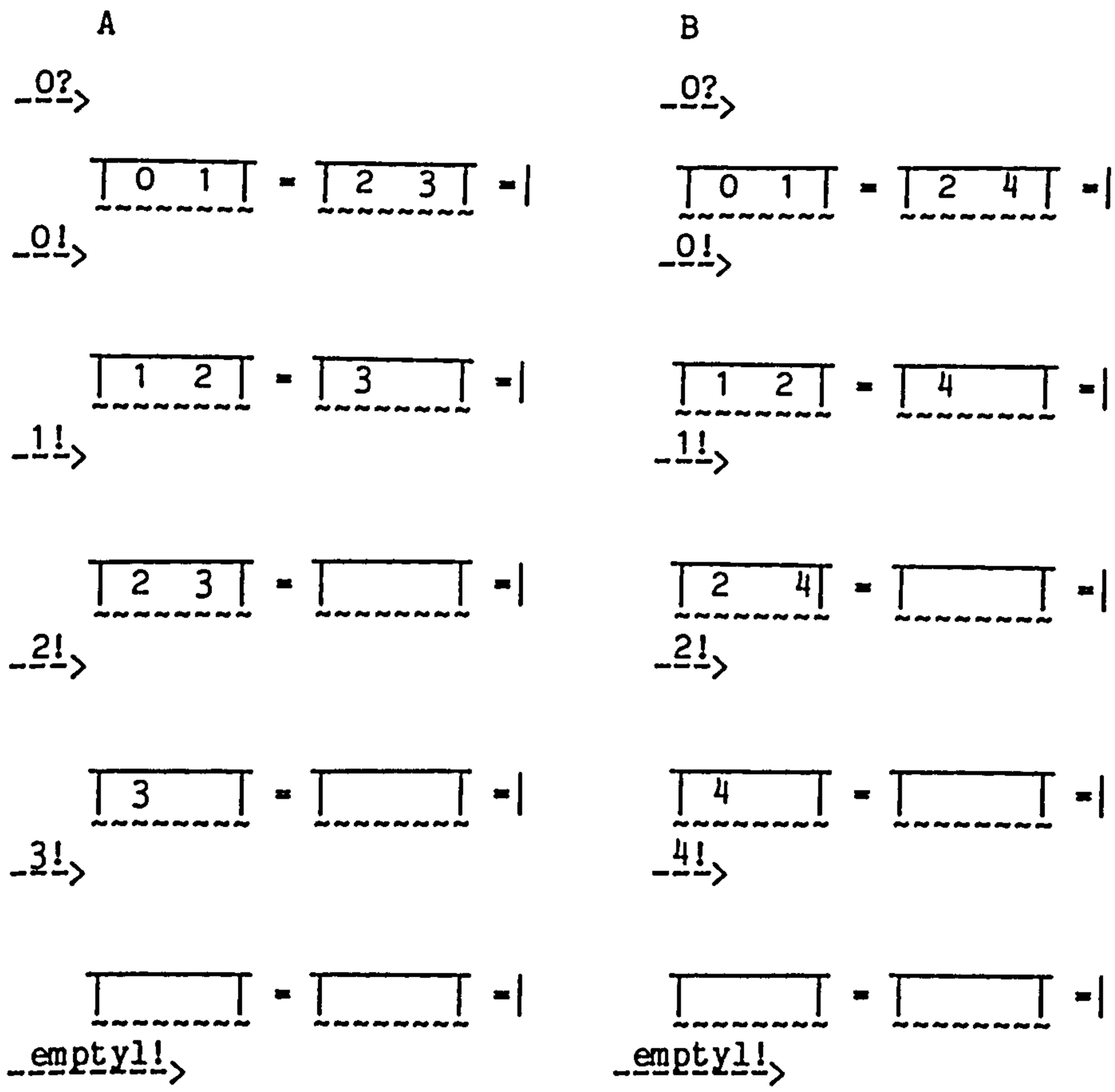
$$\mathop{\&}_{v \in M--[m]} (\text{CELL2}(m,v) \iff WS^{n-1}(M--[m,v]))$$

We expressed our indifference to the choice of v by using ⊕; this allows succinct expression of a number of machines with similar behaviour, i.e. they will all be implementations of the specification to follow. However, their behaviour can appear very different; consider (loosely):

A: $\lceil 1 \;\; 2 \rceil - \lceil 3 \;\; 4 \rceil -| \;\; \varepsilon \; ds(WS^2([1,2,3,4]))$

B: $\lceil 1 \;\; 4 \rceil - \lceil 2 \;\; 3 \rceil -| \;\; \varepsilon \; ds(WS^2([1,2,3,4]))$

If we add a new value, say 0, then the value at the right end (i.e. 4 for A, 3 for B) 'drops off':



Thus $WS^n(M)$ will describe a family of possible machines (i.e. $ds(WS^n(M))$) each with slightly different behaviour. We have chosen to abstract away from a complete-description of the

distribution of elements of the multiset M over $WS^{2n}(M)$ because such a description would (probably) depend on the exact structure of a weavesort machine and be very unwieldy. However, it can easily be seen that every distribution described (i.e. every member of $ds(WS^{2n}(M))$) can arise (the proof of implementation establishes the converse) by the following informal argument,

Claim Let $n \geq 0$, M a multiset of not more than 2n natural numbers; then any of the distributions described above of M over an n cell weavesort machine is attainable by a sequence of $|M|$ inputs to $WS^{n}(\phi)$.

Proof by induction on n and cases on M:

BASE $n = 0$: Trivial.

STEP Suppose the result is true for n-1;

case $M = \phi$: Trivial (the empty sequence).

case $M = [m]$: Just input m.

case $M = M'{+}{+}[m,v]$, $m = min(M)$: by the inductive hypothesis we have, for any distribution of M' in $WS^{n-1}(M')$, a sequence $v_1,\ldots,v_k$, $k \leq n-1$, of values such that $M' = [v_1,\ldots,v_k]$ and inputting this sequence to $WS^{n-1}(\phi)$ produces that distribution. The required sequence for the larger machine is then $m,v_1,\ldots,v_k,v$.

□

It remains to describe CELL2 more formally. A cell with two elements has two options, firstly it can accept a value from the left and then output to the right its larger value for storage by the smaller weavesort machine. It remains as a cell with two elements but now with the new smallest value (i.e. the smaller of the new value and the old smallest value) in the left register.

Alternatively, a cell with two elements can output to the left the current smallest value and then receive from the right either a value or a signal that the smaller machine is empty. If a value is received it will be (by recursive reasoning) the smallest value held by the smaller machine; comparing this with the value remaining in this leftmost cell determines the new smallest value. If the empty signal is received then the only value remaining is that in this leftmost cell.

```
CELL2(x,y) <=  l?z.r!y.SWAP(x,y)
             +
               l?x.( r?z.SWAP(y,z)
                   +
                     emptyr?.CELL1(y))
```

Note there is a slight problem here with the interpretation of CELL2 as a VLSI structure; after receipt of a new value from the left CELL2 is holding (i.e. binding in CCS) 3 values in supposedly only 2 registers. Moving to a synchronous description of the machine would solve this problem, but for our purposes it doesn't really matter.

Gathering together all these definitions we get the following definition set:

Let $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, $m = \min(M)$ if $M \neq \phi$;

$WS^n(M) <=$

```
        if n = 0 then stopper
else    if M = φ then CELL0 <=> WS^n(φ)
else    if M = [m] then CELL1(m) <=> WS^(n-1)(φ)
else    Σ_{v∈M--[m]} (CELL2(m,v) <=> WS^(n-1)(M--[m,v]))

stopper <=  l?z.stopper
          +
            empty!.stopper
```

```
CELLO          <=  l?z.CELL1(z)
                 +
                   empty1!.CELLO

CELL1(x)       <=  l?z.SWAP(x,z)
                 +
                   l!x.CELLO

CELL2(x,y)     <=  l?z.r!y.SWAP(x,z)
                 +
                   l!x.( r?z.SWAP(y,z)
                        +
                        emptyr?.CELL1(y) )

SWAP(x,y)      <=  if x < y then CELL2(x,y) else CELL2(y,x)
```

Having described a weavesort machine we now turn to the task of finding a concise expression of its intended behaviour.

Let $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, $m = \min(M)$ if $M \neq \phi$; then define a sorting machine, of capacity for 2n elements, holding M by:

$$
\begin{aligned}
SORT^{2n}(M) \Leftarrow\ & \\
& \text{if } |M| = 0 \text{ then } empty1!.SORT^{2n}(M) \\
& + \\
& \text{if } |M| > 0 \text{ then } l!m.SORT^{2n}(M--[m]) \\
& + \\
& \text{if } |M| < 2n \text{ then } l?z.SORT^{2n}(M++[z]) \\
& + \\
& \text{if } |M| = 2n \text{ then } l?z.\mathop{\Large\&}_{w \in M++[z]} SORT^{2n}((M++[z])--[w])
\end{aligned}
$$

Informally this may be read:

1. If the machine is empty it can signal that fact.

2. If the machine holds any values it should be prepared to deliver the smallest, adjusting its contents accordingly.

3. If the machine is not yet full it should be prepared to accept and hold another value.

4. If the machine is full it should still be prepared to accept values but it is conceded that, since its capacity is exceeded, some (unspecified) value must be lost from the contents.

It is the indeterminacy of this last clause which enables $SORT^{2n}(M)$ to describe every definite state of $WS^{n}(M)$; in each such process the element to be thrown away is completely determined by its distribution of values.

6.2.3. The statement of correctness

We would like some way of saying that $SORT^{2n}(M)$ is a 'good-description' of $WS^{n}(M)$ for appropriate n and M. So far we have seen three possible ways of expressing this:

a) $WS^{n}(M) \sim SORT^{2n}(M)$

b) $WS^{n}(M)$ <u>completely-implements</u> $SORT^{2n}(M)$

c) $WS^{n}(M)$ <u>implements</u> $SORT^{2n}(M)$

The point we wish to make in this subsection is that of the above only c) holds, for the following reason.

The weavesort machine WS is always willing to accept another value even if it means throwing away a value already held (or just accepted). Which value gets thrown away depends on the distribution of values in the machine, which in turn depends on the history of the machine (as was shown in the earlier Claim).

The sorting machine described by SORT is also willing to accept a value, only to throw another away if it is full. However, the

description of SORT is sufficiently perspicuous that we can see it may throw away <u>any</u> value it holds, or has just accepted, if its capacity is exceeded.

It is <u>not</u> the case, in general, that WS has license to throw way whichsoever value it chooses, so, in general again, SORT has more capabilities than WS. In fact c) will express our ignorance of <u>exactly</u> what WS will throw away, but if SORT must pass a test then so must WS, and WS cannot do anything SORT cannot do.

With a few examples we will show why a) and b) do not hold in general.

<u>Example</u> $WS^2([1,2,3,4]) \not\leftrightarrow SORT^4([1,2,3,4])$

First consider the definite states of $WS^2([1,2,3,4])$:

$ds(WS^2([1,2,3,4])) =$

$\{CELL2(1,2) \leftrightarrow CELL2(3,4) \leftrightarrow stopper,$

$CELL2(1,3) \leftrightarrow CELL2(2,4) \leftrightarrow stopper,$

$CELL2(1,4) \leftrightarrow CELL2(2,3) \leftrightarrow stopper\}$

Now, the sorting machine can accept a new value (5, say) and throw away <u>any</u> value in [1,2,3,4], so we choose 2 just to be awkward:

$SORT^4([1,2,3,4]) \xrightarrow{1?5} SORT^4([1,3,4,5])$

However

$WS^2([1,2,3,4]) \xrightarrow{1?5} p$

implies p still holds the value 2, so

$p \xrightarrow{1!1} \xrightarrow{1!2}$

but

$\not\exists\, q.\ SORT^4([1,3,4,5]) \xrightarrow{1!1} \xrightarrow{1!2} q$

Hence there can be no bisimulation containing the pair

$(WS^2([1,2,3,4]), SORT^4([1,2,3,4]))$

□

For exactly the same reasons b) is not in general true:

Example In NEXP(CCS)

$WS^2([1,2,3,4])$ completely-/implements $SORT^4([1,2,3,4])$:

As in the bisimulation example

$1!5.1?x.1?y.$if $y=2$ then $\omega!$ | $WS^2([1,2,3,4])$ must

whereas

$1!5.1?x.1?y.$if $y=2$ then $\omega!$ | $SORT^4([1,2,3,4])$ mu/st

□

It only remains to show why c) is an appropriate way of saying that $SORT^{2n}(M)$ is a 'good description' of $WS^n(M)$. The idea is that the behaviour of the machines differs only when an overflow occurs, i.e. too many values have been input. The weavesort machine will throw away a value determined completely, but in a non-trivial way, by its initial distribution of values and its subsequent interactions. The abstract sorting machine can throw away any of the values it holds. One of the values the sorting machine could throw away is the value actually thrown away by the weavesort machine. Thus if we are given the definition of $SORT^{2n}(M)$ and told

$$WS^n(M) \text{ implements } SORT^{2n}(M)$$

then we can deduce when overflow occurs but not exactly which value is lost.

6.3. Proving implementation

In this section we prove that $\forall\ n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$.

$WS^n(M)$ <u>implements</u> $SORT^{2n}(M)$

that is $\forall\ t \in T$.

$t[WS^n(M)]$ <u>may</u> implies $t[SORT^{2n}(M)]$ <u>may</u>,

$t[SORT^{2n}(M)]$ <u>must</u> implies $t[WS^n(M)]$ <u>must</u>

We have already seen in the preceding section that $SORT^{2n}$ has more capabilities than WS^n so the relationship of complete-implementation or of having a bisimulation does not hold. The most we can hope for is implementation, and that is what we shall prove.

6.3.1. Proving the may part

In this subsection we aim to prove (loosely)

$t[WS^n(M)]$ <u>may</u> implies $t[SORT^{2n}(M)]$ <u>may</u>

and we achieve this essentially by noting that if $t[WS^n(M)]$ <u>may</u> then it is because $WS^n(M)$ can perform some sequence of actions which may satisfy t, and any process which can perform this sequence can similarly satisfy t. We accordingly introduce the notion of simulation wherein if p_1 <u>simulates</u> p_2 then any sequence of (external) actions performable by p_1 is performable by p_2 also. Demonstrating that $WS^n(M)$ <u>simulates</u> $SORT^{2n}(M)$ then completes the proof.

We begin by employing a strong version of half the bisimulation relation in the notion of 'simulation', whence we can employ something similar to half of the bisimulation proof technique.

The basic idea is to assume that the process being simulated has no silent actions, as is often reasonable when it is a specification.

Definition 6.9

A relation S $\subseteq$ Terms x Terms is a <u>simulation</u> if whenever p S q,

1. $p \xrightarrow{1} p'$ implies p' S q

2. $p \xrightarrow{a} p'$, $a \neq 1$ implies $\exists\, q'.\ q \xrightarrow{a} q'$, p' S q'

If S is a simulation and p S q then we say p <u>simulates</u> q.

□

We would now like to slightly extend Definition 6.9, via the following proposition, so as to remove the need for pairs of the form $(WS^n(M),p)$ in the simulation.

Proposition 6.5

$(\forall\, dp \in ds(p).\ dp$ <u>simulates</u> $q)$ iff p <u>simulates</u> q.

Proof Suppose $dp \in ds(p)$, p S q, S a simulation, then take

$S' = S + \{(dp,q)\}$

Since S is a simulation we need only examine the pair (dp,q)

<u>case</u> $dp \xrightarrow{1} p'$: then by Proposition 6.1

$p \xrightarrow{1} p'$

so

p' S q.

<u>case</u> $dp \xrightarrow{a} p'$, $a \neq 1$: then by Proposition 6.1

$p \xrightarrow{a} p'$, $\exists\, q'.\ q \xrightarrow{a} q'$, p' S q'.

The converse is proved in a similar manner.

□

Proposition 6.6

If t[p] -> t'[p'], p <u>simulates</u> q

then ds(t') = {t'} and one of the following holds:

1. t[q] -> t'[q], p' <u>simulates</u> q

2. t' ε ds(t), p $-\underline{1}\rightarrow$ p'

3. ∃ q'. t[q] -> t'[q'], p' <u>simulates</u> q'

Proof we proceed by cases on Proposition 6.4 (4),

<u>case</u> t[.] $-\underline{1}\rightarrow$ t'[.], p' ε ds(p): then by Proposition 6.4 (2)

and Proposition 6.3

$t[q] -\underline{1}\rightarrow t'[q]$

and by Proposition 6.5

p' <u>simulates</u> q.

<u>case</u> t[.] $-[\underline{1}]\rightarrow$ t'[.], p $-\underline{1}\rightarrow$ p': then by Proposition 6.4 (2)

t' ε ds(t).

<u>case</u> t[.] $-[\underline{a}]\rightarrow$ t'[.], p $-\underline{a}\rightarrow$ p', a ≠ 1: then

∃ q'. q $-\underline{a}\rightarrow$ q', p' <u>simulates</u> q'

Hence by Definition 6.3 and Proposition 6.4 (4)

∀ dq' ε ds(q'). t[q] $-\underline{1}\rightarrow$ t'[dq']

so by Definition 6.3 and Proposition 6.3

$t[q] -\underline{1}\rightarrow t'[q']$

since

t' = ds(t')

□

Corollary 6.1

If t[p] -> => t'[p'], p <u>simulates</u> q

then ∃ t", q'. t' ε ds(t"), t[q] ->* t"[q'], p' <u>simulates</u> q'

Proof by induction on the length n of the derivation '=>',

<u>case</u> n = 0: Immediate by Proposition 6.6.

<u>case</u> n > 0: then

$\exists\ t_1, p_1.\ t[p] \rightarrow\!\!* t_1[p_1] \rightarrow t'[p']$

so

$\exists\ t_2, q_1.\ t[q] \rightarrow\!\!* t_2[q_1],\ t_1 \in ds(t_2),\ p_1$ *simulates* q_1

By Proposition 6.6 we have 3 cases;

case $t_1[q_1] \xrightarrow{1} t'[q_1],\ ds(t') = \{t'\},\ p'$ *simulates* q_1:

So by Proposition 6.1

$t_2[q_1] \rightarrow t'[q_1]$

case $t' \in ds(t_1),\ p_1 \xrightarrow{1} p'$: then by Definition 6.9

p' *simulates* q_1

and

$t' = t_1 \in ds(t_2)$

since

$ds(t_1) = \{t_1\}$.

case $\exists\ q'.\ t_1[q_1] \xrightarrow{1} t'[q'],\ p'$ *simulates* q':

then by Proposition 6.1 and Proposition 6.3

$t_2[q_1] \xrightarrow{1} t'[q']$.

□

Now we can state and prove the principle proposition by which we establish $t[WS^n(M)]$ *may* implies $t[SORT^{2n}(M)]$ *may*

Proposition 6.7

p *simulates* q implies $\forall\ t \in T.\ (t[p]$ *may* implies $t[q]$ *may*$)$

Proof Suppose p *simulates* q and $t[p]$ *may*, then by Definition 4.8 and Proposition 6.4 there is some successful computation

$\langle t_k[p_k]\rangle_K$

such that if $K = \{0\}$ then

$t[p] = t_0[p_0] \rightarrow t_1[p_1] \rightarrow \ldots$

and if $K = \{0\}$ then we have a computation of length 0 and, by our

convention and Proposition 6.3,

$t_0 \in ds(t)$, $p_0 \in ds(p)$

<u>case</u> K = {0}: then by Proposition 6.4 and Definition 6.5

$t_0[.] \xrightarrow{\omega!}$

so by Proposition 6.1 and Proposition 6.3

$t[.] \xrightarrow{\omega!}$

Now we have two cases depending on the computations of t[q]:

<u>case</u> $t[q] \rightarrow$: then t[q] is the first configuration in some computation, and by Proposition 6.4 $t[q] \xrightarrow{\omega!}$.

<u>case</u> $t[q] \not\rightarrow$: then we have a computation of zero length, but by Proposition 6.3 and Proposition 6.4 (2)

$\forall\, q_0 \in ds(q).\ t_0[q_0] \in ds(t[q]),\ t_0[q_0] \xrightarrow{\omega!}$

<u>case</u> K = {0} : then

$\exists\, j \geq 1.\ t[p] \rightarrow \Rightarrow t_j[p_j] \xrightarrow{\omega!}$

so by Corollary 6.1

$\exists\, q', t''.\ t[q] \xrightarrow{1}{}^{*}\ t''[q'],\ t_j \in ds(t'')$,

whence by Proposition 6.1 and Proposition 6.2

$\exists\, r.\ t[q] \xrightarrow{1}{}^{*}\ r \xrightarrow{\omega!}$

□

The next step is now clear, we need to prove that for each $n \geq 0$ and M $\in$ multisets(Nat), $|M| < 2n$

$\forall\, dp \in ds(WS^n(M)).\ WS^n(M)$ <u>simulates</u> $SORT^{2n}(M)$

We do this by defining for each $n \geq 0$ the required simulation S_n for weavesort machines of n cells and any M, $|M| \leq 2n$, by induction on n:

Definition 6.10

<u>BASE</u> n=0:

$S_0 = \{(\text{stopper}, \text{SORT}^0(\phi))\}$

STEP: Suppose we have constructed S_{n-1}, define S_n to be the union of the following sets

[1] $\{(\text{CELLO} \leftrightarrow p\ ,\ \text{SORT}^{2n}(\phi)):$
$p\ S_{n-1}\ \text{SORT}^{2(n-1)}(\phi)\}$

[2] $\{(\text{CELL1}(x) \leftrightarrow p\ ,\ \text{SORT}^{2n}([x])):$
$p\ S_{n-1}\ \text{SORT}^{2(n-1)}(\phi)\}$

[3] $\{(\text{CELL2}(m,v) \leftrightarrow p\ ,\ \mathop{\Large\mathbb{Q}}_{N \in Ns} \text{SORT}^{2n}(N)):$
$p\ S_{n-1}\ \mathop{\Large\mathbb{Q}}_{M \in Ms} \text{SORT}^{2(n-1)}(M -\!- [m,v]),$
$Ms \subseteq Ns \subseteq \{M: |M| \leq 2n, [m,v] \subseteq M, m = \min(M)\}\}$

[4] $\{(r!v.\text{SWAP}(m,v') \leftrightarrow p\ ,\ \mathop{\Large\mathbb{Q}}_{N \in Ns} \text{SORT}^{2n}(N{+}{+}[v'])):$
$p\ S_{n-1}\ \mathop{\Large\mathbb{Q}}_{M \in Ms} \text{SORT}^{2(n-1)}(M -\!- [m,v]),$
$Ms \subseteq Ns \subseteq \{M: |M| < 2n, [m,v] \subseteq M, m = \min(M)\}\}$

[5] $\{(r!v.\text{SWAP}(m,v') \leftrightarrow p\ ,\ \mathop{\Large\mathbb{Q}}_{N \in NsR} \text{SORT}^{2n}(N)):$
$p\ S_{n-1}\ \mathop{\Large\mathbb{Q}}_{M \in Ms} \text{SORT}^{2(n-1)}(M -\!- [m,v]),$
$Ms \subseteq Ns \subseteq \{M: |M| = 2n, [m,v] \subseteq M, m = \min(M)\},$
$NsR = \{(N{+}{+}[v'])-\!-[w]: N \in Ns, w \in N{+}{+}[v']\}\}$

[6] $\{(r?x.\text{SWAP}(v,x){+}\text{empty}r?.\text{CELL1}(v)) \leftrightarrow p\ ,\ \mathop{\Large\mathbb{Q}}_{N \in Ns} \text{SORT}^{2n}(N)):$
$p\ S_{n-1}\ \mathop{\Large\mathbb{Q}}_{M \in Ms} \text{SORT}^{2(n-1)}(M -\!- [v]),$
$Ms \subseteq Ns \subseteq \{M: |M| < 2n, [v] \subseteq M\}\}$

□

Note if Val were allowed to be infinite then $\mathop{\Large\mathbb{Q}}_{N \in Ns} \text{SORT}^{2n}(N)$ could be an infinite indeterminate summation in many of the above cases.

The motivations behind this seemingly complex definition are quite

simple, regarding a n+1 cell weavesort machine as a single cell on the front of the SORT machine of size 2(n-1). We motivate $S_n[3]$ and $S_n[5]$ by considering a simplified first attempt at both:

$$[3.1]\ \{(CELL2(m,v) <=> p,\ SORT^{2n}(M)):$$
$$p\ S_{n-1}\ SORT^{2(n-1)}(M--[m,v]),$$
$$M \in \{M:\ |M| < 2n,\ [m,v] \subseteq M,\ m = min(M)\}\}$$

$$[5.1]\ \{(r!v.SWAP(m,v') <=> p,\ \mathop{\Phi}_{w \in M++[v']} SORT^{2n}(M++[v']--[w])):$$
$$p\ S_{n-1}\ SORT^{2(n-1)}(M--[m,v]),$$
$$M \in \{M:\ |M| = 2n,\ [m,v] \subseteq M,\ m = min(M)\}\}$$

For example $S_n[5.1]$ might (loosely) be read;

Suppose M ε multisets(Nat), |M| = 2n, [m,v] ⊆ M, m = min(M), and p *simulates* $SORT^{2(n-1)}(M--[m,v])$, then r!v.SWAP(m,v') <=> p *simulates* $\mathop{\Phi}_{w \in M++[v']} SORT^{2n}(M++[v']--[w])$

This is essentially because $SORT^{2(n-1)}(M--[m,v])$ has put an 'upper bound' on what p can do, so we can substitute $SORT^{2(n-1)}(M--[m,v])$ for p to get an upper bound on what r!v.SWAP(m,v')<=>p can do.

So now we see the first problem; the occurrences of '*simulates*' above should be replaced by S_{n-1} in the first instance and S_n in the second. This means that what will be the inductive hypothesis is of the form

$$q\ S_{n-1}\ SORT^{2(n-1)}(N)$$

while the consequent is of the form

$$q'\ S_n\ \mathop{\Phi}_{N \in Ns}\ SORT^{2n}(N)$$

Thus we need a slightly weaker hypothesis, replacing [5.1] by

[5.2] $\{(r!v.SWAP(m,v') <=> p, \mathop{\Phi}_{M\varepsilon MsR} SORT^{2n}(M)):$

$p\ S_{n-1}\ \mathop{\Phi}_{M\varepsilon Ms} SORT^{2(n-1)}(M--[m,v]),$

$Ms\ \underline{C}\ \{M:\ |M| = 2n,\ [m,v]\ \underline{C}\ M,\ m = min(M)\},$

$MsR = \{M++[v']--[w]:\ M\ \varepsilon\ Ms,\ w\ \varepsilon\ M++[v']\}\}$

Note we have rewritten

$$\mathop{\Phi}_{M\varepsilon Ms}\ \mathop{\Phi}_{w\varepsilon M++[v']} SORT^{2n}(M++[v']--[w])$$

as

$$\mathop{\Phi}_{M\varepsilon MsR} SORT^{2n}(M)$$

so that Proposition 6.10 (2.1) will hold.

The second problem arises when considering the case

$$r!v.SWAP(m,v') <=> p \xrightarrow{1} q,\ p \xrightarrow{1?v} p',\ q = SWAP(m,v') <=> p'$$

then by Proposition 6.11 and the inductive hypothesis

$$p'\ S_{n-1}\ \mathop{\Phi}_{M'\varepsilon MsR'} SORT^{2n}(M'), \qquad (1)$$

$$MsR\ \underline{C}\ \{M--[m,w]:\ M\ \varepsilon\ Ms,\ w\ \varepsilon\ M--[m]\}$$

However, we want to use $S_n[3.1]$. If we used (1) in $S_n[3.1]$ then we would get

$$q\ S_n[3.1]\ \mathop{\Phi}_{M''\varepsilon MsR''} SORT^{2n}(M'')$$

where

$$MsR'' = \{M++[v']--[w]:\ M\ \varepsilon\ Ms,\ w\ \varepsilon\ M--[m]\}$$

$$\underline{C}\ MsR$$

and in general MsR" may be a proper subset of MsR. What has happened?

The problem is that we have gained information about which values can be thrown away (viz. we can't throw away m or v') but this gain of information can't be expressed in even a silent transition of the indeterminate SORT machine. To get around this we draw a veil over our information gain, essentially by noting that

$\forall\ q_1, q_2, r.\ q_1$ <u>simulates</u> q_2 implies q_1 <u>simulates</u> $q_2 \oplus r$

By restricting the allowable forms of r, for our purposes, to suitable sorting machines we can use a set NsR, bounded above by conditions of suitability and below by conditions of necessity, to give the desired leeway. This is the purpose of the set NsR in [3] and [5]; to see how it works in practice this example is covered formally in the proof.

For each n, S_n has some simple properties which will be useful in proving S_n to be a simulation,

Proposition 6.10

Let $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, then

1. $\forall\ pp \in ds(WS^n(M)).\ pp\ S_n\ SORT^{2n}(M)$
2. If $pp\ S_n\ q$ then $\exists$ finite Ms $\subseteq$ multisets(Nat) such that
   1. $q = \bigoplus_{M \in Ms} SORT^{2n}(M)$
   2. $\forall\ M, M' \in Ms.\ |M| = |M'|$

Proof by induction on n.

□

Hereafter we allow Ms to range only over finite subsets of multisets(Nat) and write |Ms| when referring to the cardinality of the members of Ms rather than the cardinality of Ms itself.

From Proposition 6.10 (2) we can see that we will be interested in transitions between particular forms of indeterminate terms. We need some way of employing the definition of SORT to generate these transitions,

Proposition 6.11

Suppose $n \geq 0$, and Ms,Ns $\subseteq$ multisets(Nat), then

$$\mathop{\mathbb{Q}}_{M \in Ms} SORT^{2n}(M) \xrightarrow{a} \mathop{\mathbb{Q}}_{N \in Ns} SORT^{2n}(N)$$

iff

$$Ns \subseteq \{N: \} M \in Ms.\ SORT^{2n}(M) \xrightarrow{a} SORT^{2n}(N)\}$$

Proof Immediate by Definition 6.3.

□

Proposition 6.12

Suppose $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, then

$WS^n(M)$ <u>simulates</u> $SORT^{2n}(M)$

Proof We will prove

$$\forall\ pp \in ds(WS^n(M)).\ pp \text{ } \underline{\text{simulates}} \text{ } SORT^{2n}(M)$$

and we do this by showing

1. $\forall\ pp \in ds(WS^n(M)).\ pp\ S_n\ SORT^{2n}(M)$
2. S_n is a simulation.

Part 1 has already been shown in Proposition 6.10. Part 2 most properly proceeds by induction on n but, as with bisimulations, the great amount of case analysis and duplication in the proof dictates that, in this presentation, we consider two exemplars only; pairs of the forms given in $S_n[3]$ and $S_n[5]$.

We begin with $S_n[3]$, so assume S_{n-1} is a simulation,

$$(CELL2(m,v) \Leftrightarrow p\ ,\ \mathop{\mathbb{Q}}_{M \in Ms} SORT^{2n}(M)) \in S_n[3]$$

and

$$CELL2(m,v) \Leftrightarrow p \xrightarrow{a} r\ .$$

We proceed by cases on the derivation:

<u>case</u> a=1, r = $CELL2(m,v) \Leftrightarrow p'$, $p \xrightarrow{1} p'$:

From the inductive hypothesis,

$$p'\ S_{n-1}\ \mathop{\mathbb{Q}}_{M \in Ms} SORT^{2n-1}(M--[m,v])$$

Hence,

$r\ S_n[3]\ \oplus_{N \varepsilon Ns} SORT^{2n}(N)$

case a=1!m, r = (r?z.SWAP(v,z)+emptyr?.CELL1(v)) <=> p: then

$r\ S_n[6]\ \oplus_{N' \varepsilon Ns'} SORT^{2n}(N')$

where

$Ns' = \{N--[m]: N \varepsilon Ns\}$

By Proposition 6.11

$\oplus_{N \varepsilon Ns} SORT^{2n}(N) \xrightarrow{1!m} \oplus_{N' \varepsilon Ns'} SORT^{2n}(N')$

case a=1?v', r = r!v.SWAP(m,v') <=> p:

We have two cases depending on whether or not the submachine p will overflow upon receipt of the value v,

case |Ms| < 2n: then the submachine won't overflow and

$r\ S_n[4]\ \oplus_{N \varepsilon Ns} SORT^{2n}(N++[v'])$

By Proposition 6.11

$\oplus_{N \varepsilon Ns} SORT^{2n}(N) \xrightarrow{1?v'} \oplus_{N \varepsilon Ns} SORT^{2n}(N++[v'])$

case |Ms| = 2n: then the submachine will overflow and

$r\ S_n[5]\ \oplus_{N' \varepsilon Ns'} SORT^{2n}(N')$

where

$Ns' = \{(N++[v'])--[w]: N \varepsilon Ns, w \varepsilon N++[v']\}$

By Proposition 6.11

$\oplus_{N \varepsilon Ns} SORT^{2n}(N) \xrightarrow{1?v'} \oplus_{N' \varepsilon Ns'} SORT^{2n}(N')$

That concludes the proof for $S_n[3]$, so now we turn to $S_n[5]$; suppose S_{n-1} is a simulation and

$(r!v.SWAP(m,v') <=> p\ ,\ \oplus_{N \varepsilon NsR} SORT^{2n}(N)) \varepsilon S_n[5]$

and

$r!v.SWAP(m,v') <=> p \xrightarrow{a} r$

We proceed by cases on the derivation

case a=1, r = SWAP(m,v') <=> p', p $\xrightarrow{1?v}$ p':

From the inductive hypothesis, Proposition 6.10 (2.1),

Proposition 6.11 and the definition of SORT we have

$$p' \; S_{n-1} \; \mathop{\mathbb{Q}}_{M \in Ms \, sR} \; SORT^{2n}(M'),$$

$$MsR \subseteq \{M--[m,w]: M \in Ms, \; w \in M--[m]\}$$

$$= \{(M++[v']--[w])--[m,v']: M \in Ms, \; w \in M--[m]\}$$

$$\subseteq \{(N++[v']--[w])--[m,v']: N \in Ns, \; w \in N--[m]\}$$

$$\subseteq \{(N++[v']--[w])--[m,v']: N \in Ns, \; w \in N++[v']\}$$

$$= \{Nr--[m,v']: Nr \in NsR\}$$

thus

$$r \; S_n[5] \; \mathop{\mathbb{Q}}_{N \in NsR} SORT^{2n}(N)$$

case a=1, $r = r!v.SWAP(m,v') \Leftrightarrow p', \; p \xrightarrow{1} p'$:

From the inductive hypothesis, Proposition 6.10 (2.1), Proposition 6.11 and the definition of SORT we have

$$p' \; S_{n-1} \; \mathop{\mathbb{Q}}_{M \in Ms} SORT^{2(n-1)}(M--[m,v])$$

so

$$r \; S_n[5] \; \mathop{\mathbb{Q}}_{N \in NsR} SORT^{2n}(N)$$

□

Corollary 6.2

Suppose $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, then

$$\forall \; t \in T. \; (t[WS^n(M)] \; \underline{may} \; \text{implies} \; t[SORT^{2n}(M)] \; \underline{may})$$

□

6.3.2. Proving the must part

In this section we aim to prove (loosely)

$$t[SORT^{2n}(M)] \; \underline{must} \; \text{implies} \; t[WS^n(M)] \; \underline{must}$$

To see how we will do this consider for the moment the case $n>0$, $[m,v] \subseteq M$, $m = min(M)$; the proof has three steps:

1. We will show

$$\forall\, v \in M\text{--}[m].\ t[CELL2(m,v) \Leftrightarrow SORT^{2(n-1)}(M\text{--}[m,v])] \ \underline{must}$$

implies

$$t[WS^{n}(M)]\ \underline{must}$$

that is we can substitute the $SORT^{2(n-1)}$ specification for the WS^{n-1} implementation in every possible experiment. This is achieved essentially by regarding the front cell together with t as a test t_v

$$t_v[.] = t[\ CELL2(m,v) \Leftrightarrow [.]\ ]$$

on the smaller machine WS^{n-1}.

2. We prove the antecedent in 1. by showing that for any $k \geq 0$,

$$t[SORT^{2n}(M)]\ \underline{must}_k$$

implies

$$\forall\, v \in M\text{--}[m].\ t_v[SORT^{2(n-1)}(M\text{--}[m,v])]\ \underline{must}_{2k}$$

where we define for $k \geq 0$

$e\ \underline{must}_k$ if within k transition steps e must succeed

whence

$e\ \underline{must}_k$ implies $e\ \underline{must}$

This is achieved via the notion of a 'ranking', allowing a bisimulation-style proof technique.

3. We establish the antecedent in 2. by showing

$$t[SORT^{2n}(M)]\ \underline{must}$$

implies

$$\exists\, k \geq 0.\ t[SORT^{2n}(M)]\ \underline{must}_k$$

To show this we need to restrict Val, the set of values passable in a communication (chapter 1 section 2) to be a finite set, then we are considering finitely branching computation trees and we can apply Konigs Lemma.

Having described the proof 'top-down' we present it bottom-up, thus the order of presentation is 3, 2, and then 1.

6.3.2.1. Proving part 3

As in Chapter 5 the proof of part 3 is motivated by considering the computation trees of experiments, regarding the set of computations of an experiment e as a tree 'comptree(e)'. Suppose e is an experiment and e must, then every computation (i.e. maximal path through the tree) must begin with a finite derivation leading to a successful configuration. Now suppose we have a finitely branching computation tree T and consider the tree T' with every sub-tree of T whose root is a successful configuration replaced by some distinguished leaf. Then T' must be finite (by Konigs Lemma) since all the derivations are finite. Hence the tree T' is of finite depth $k \geq 0$, so e must succeed within k steps, which we formalise in the following predicate:

Definition 6.11

Define the family of predicates $\underline{must}_k$ $k \geq 0$ by

e $\underline{must}_0$ iff success(e)

e $\underline{must}_{k+1}$ iff

either 1. success(e)

or 2. $\exists$ e'. e -> e',

$$\forall\ e'.\ e \rightarrow e' \quad \text{implies} \quad e'\ \underline{must}_k$$

□

Now we formally state, but don't formally prove here, the upshot of the above informal argument,

Proposition 6.13

Suppose e is an experiment, e $\underline{must}$, and comptree(e) is finitely branching, then $\exists\ k \geq 0.\ e\ \underline{must}_k$

□.

It now remains to prove that the experiments we are interested in have finitely branching computation trees,

Proposition 6.14

Suppose p is a term and we restrict Val (the set of values passable in a communication) to be finite, then comptree(p) is finitely branching.

Proof by structural induction on p; the restriction on Val is required for terms a.p'. Note we have not allowed infinite summation (either + or ⊕) in the construction of terms.

□

From this point on we consider Val to be a finite set, and so

Corollary 6.3

$\forall\ n \geq 0,\ M \in \text{multisets(Nat)},\ |M| \leq 2n,\ t \in T.$

$$t[SORT^{2n}(M)]\ \underline{must} \quad \text{implies} \quad \exists\ k \geq 0.\ t[SORT^{2n}(M)]\ \underline{must}_k$$

□

This concludes part 3 of the proof.

6.3.2.2. Proving part 2

We will now employ the results of part 3 in the following fashion,

Proposition 6.15

Suppose M ε multisets(Nat), |M| ≤ 2n, t ε T, and

$t[SORT^{2n}(M)]\ \underline{must}_k$

for some k ≥ 0, then

1. If M = φ then

$t[CELL0 <=> SORT^{2(n-1)}(\phi)]\ \underline{must}_{2k}$

2. If M = [m] then

$t[CELL1(m) <=> SORT^{2(n-1)}(\phi)]\ \underline{must}_{2k}$

3. If 2 ≤ |M| ≤ 2n, m = min(M) then

$\forall\ v \in M--[m].\ t[CELL2(m,v) <=> SORT^{2(n-1)}(M--[m,v])]\ \underline{must}_{2k}$

□

The factor of two difference between the $\underline{must}_k$ of the antecedent and the $\underline{must}_{2k}$ of the consequents arises because a cell holding two values must communicate with both the user and the SORT machine of size n-1 in order to model the single communication between the user and the SORT machine of size n. Thus if we were to replace the smaller SORT machine by a smaller weavesort machine we would get another n-1 factors of 2, but that doesn't concern us here.

To prove this proposition we introduce the notion of 'rankings', closely mirroring the definition of $\underline{must}_k$.

Definition 6.12

A <u>ranking</u> of experiments is an indexed family of sets $\langle R_k \rangle_K$ such that

1. $O \in K \subseteq Nat$

2. All $k \in K$. $R_k \subseteq Terms$

3. $q \in R_O$ implies success(q)

4. If $j \in K$, $q \in R_j$ then

   either 1. success(q)

   or 2. $\exists\, q'.\ q \rightarrow q'$,

   $\forall\, q'.\ (q \rightarrow q'$ implies $\exists\, i \in K.\ i < j,\ q' \in R_i)$

□

Our intent in this definition is to emulate the bisimulation proof technique; to show e $\underline{must}_j$, $j \geq O$, (and thereby e $\underline{must}$) we merely need provide a ranking $\langle R_k \rangle_K$ such that $e \in R_j$, $j \in K$. Then, by 4, either it is already successful or it can progress and every possible progression brings it closer to success. This is formalised in the following proposition,

Proposition 6.16

Suppose $\langle R_k \rangle_K$ is a ranking, $j \in K$, $q \in R_j$, then

1. q $\underline{must}_j$.

2. $\forall\, dq \in ds(q).$ dq $\underline{must}_j$

Proof by induction on $j \in K$.

□

So now, in trying to prove Proposition 6.15, we will construct a ranking $\langle R_k \rangle_K$ by defining, for each $i \geq O$, R_{2i} and R_{2i+1} as follows;

Definition 6.13

Suppose $i \geq O$, then define:

R_{2i} is the union of the following three sets:

[1] {t[CELLO <=> $SORT^{2(n-1)}(\phi)$]:

$t[SORT^{2n}(\phi)]$ $\underline{must}_i$}

[2] {t[CELL1(m) <=> $SORT^{2(n-1)}(\phi)$]:

$t[SORT^{2n}([m])]$ $\underline{must}_i$}

[3] {t[CELL2(m,v) <=> $SORT^{2(n-1)}(M--[m,v])$]:

$t[SORT^{2n}(M)]$ $\underline{must}_i$,

$[m,v] \underline{C} M$, $m = min(M)$, $|M| \leq 2n$}

R_{2i+1} is the union of the following three sets:

[4] {t[r!v.SWAP(m,v') <=> $SORT^{2(n-1)}(M--[m,v])$]:

$t[SORT^{2n}(M++[v'])]$ $\underline{must}_i$,

$[m,v] \underline{C} M$, $m = min(M)$, $|M| < 2n$ }

[5] {t[r!v.SWAP(m,v') <=> $SORT^{2(n-1)}(M--[m,v])$]:

$\forall w \in M++[v']$. $t[SORT^{2n}(M++[v']--[w])]$ $\underline{must}_i$

$[m,v] \underline{C} M$, $m = min(M)$, $|M| = 2n$ }

[6] {t[(r?z.SWAP(v,z)+emptyr?.CELL1(v)) <=> $SORT^{2(n-1)}(M--[m,v])$]:

$t[SORT^{2n}(M--[m])]$ $\underline{must}_i$,

$[m,v] \underline{C} M$, $m = min(M)$, $|M| \leq 2n$ }

□

Proposition 6.17

$\langle R_i \rangle_{Nat}$ is a ranking.

Proof we prove each of the conditions of Definition 6.12;

1. Trivial.

2. Trivial.

3. Firstly we note we are dealing with the case i=0. We will just deal with $R_0[1]$ since all cases are exactly the same; suppose

$$q = t[CELLO \leq\!\!> SORT^{2(n-1)}(\phi)] \in R_0[1]$$

and

$$t[SORT^{2n}(\phi)] \ \underline{must}_0$$

thus by Proposition 6.4 and Definition 6.11

$$t[.] \xrightarrow{\underline{\omega!}}$$

so, again by Proposition 6.4

$$q \xrightarrow{\underline{\omega!}}, \text{ i.e. success}(q)$$

4. As with a bisimulation proof there is a great deal of case analysis so, for the purposes of this presentation, we take two exemplars;

   <u>case</u> [3] Suppose

   $$q = t[CELL2(m,v) \leq\!\!> SORT^{2(n-1)}(M--[m,v])] \in R_{2i}[3]$$

   then, following the definition of a ranking, either success(q), in which case there is nothing further to prove, or by Proposition 6.4

   $$t[.] \xrightarrow{\underline{\omega!}}\!\!\!\!\!\!\not\ \ >$$

   so succ/ess($t[SORT^{2n}(M)]$). We do know, however,

   $$t[SORT^{2n}(M)] \ \underline{must}_i$$

   and hence by Proposition 6.4, Proposition 6.11, and definitions of SORT and $\underline{must}_i$ we have some of the following (i.e. at least one):

   1. $t[.] \xrightarrow{\underline{1}} t'[.]$, $t'[SORT^{2n}(M)] \ \underline{must}_{i-1}$
   2. $|M| < 2n$, $t[.] \xrightarrow{\underline{[1?v']}} t'[.]$, $t'[SORT^{2n}(M++[v'])] \ \underline{must}_{i-1}$,
   3. $|M| = 2n$, $t[.] \xrightarrow{\underline{[1?v']}} t'[.]$,

      $\forall\ w \in M++[v']$. $t'[SORT^{2n}(M++[v']--[w])] \ \underline{must}_{i-1}$
   4. $t[.] \xrightarrow{\underline{[1!m]}} t'[.]$, $t'[SORT^{2n}(M--[m])] \ \underline{must}_{i-1}$

   We now show how the above cases correspond exactly with derivations of q, i.e. if any of the above hold then q has a

corresponding derivative and any derivative of q is further down the ranking. We proceed by cases on all the possible derivatives of q, using Proposition 6.4, Proposition 6.11 and the definitions of CELL2 and SORT; we have q -> q' iff one of the following cases holds:

<u>case</u> $t[.] \xrightarrow{1} t'[.]$,

$q' = t'[CELL2(m,v) \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]$:

if this case holds then 1. above does, so we have

$t'[SORT^{2n}(M)]$ $\underline{must}_{i-1}$

and hence

$q \rightarrow q' \in R_{2(i-1)}[3]$

<u>case</u> $|M| < 2n$, $t[.] \xrightarrow{[1?v']} t'[.]$,

$q' = t'[?!v.SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]$:

if this case holds then 2. above does, so

$t'[SORT^{2n}(M++[v'])]$ $\underline{must}_{i-1}$

and hence

$q \rightarrow q' \in R_{2i-1}[4]$

<u>case</u> $|M| = 2n$, $t[.] \xrightarrow{[1?v']} t'[.]$,

$q' = t'[?!v.SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]$:

if this case holds then 3. above does, so

$\forall\ w \in M++[v'].\ t'[SORT^{2n}(M++[v']--[w])]$ $\underline{must}_{i-1}$

and hence

$q \rightarrow q' \in R_{2i-1}[5]$

<u>case</u> $t[.] \xrightarrow{[1!m]} t'[.]$,

$q' = t'[(\_) \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]$:

if this case holds then 4. above does, so

$t'[SORT^{2n}(M--[m])]$ $\underline{must}_{i-1}$

and hence

$$q \rightarrow q' \in R_{2i-1}[6]$$

That concludes the case for pairs from $R_{2i}[3]$, the other exemplar concerns pairs from $R_{2i+1}[5]$;

<u>case</u> [5] Suppose

$$q = t[?!v.SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,v])] \in R_{2i+1}[5]$$

then, following the definition of a ranking, either success(q) in which case there is nothing further to prove, or by Proposition 6.4

$$t[.] \xrightarrow{\omega!} \not\rightarrow$$

so succ/ess($t[SORT^{2n}(M)]$) which we now assume. We proceed by cases on the (non-empty) set of possible derivatives q' of q, using Proposition 6.4, Proposition 6.11 and the definitions of SORT and CELL2:

<u>case</u> $t[.] \xrightarrow{1} t'[.]$,

$$q' = t'[?!v.SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]:$$

then $\forall\ w \in M{+}{+}[v']$.

$$t[SORT^{2n}(M{+}{+}[v']--[w])] \rightarrow t'[SORT^{2n}(M{+}{+}[v']--[w])]\ \underline{must}_{i-1}$$

so

$$q \rightarrow q' \in R_{2i-1}[5]$$

<u>case</u> $t[.] \xrightarrow{[1]} t'[.]$, $t' \in ds(t)$,

$$\exists\ w \in M--[m].\ q' = t'[SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,w])]:$$

But we have

$$\forall\ w \in M{+}{+}[v'].\ t[SORT^{2n}(M{+}{+}[v']--[w])]\ \underline{must}_i$$

so by Proposition 6.3 and Proposition 6.16 (2)

$$\forall\ w \in M{+}{+}[v'].\ \forall\ t' \in ds(t).$$

$$t'[SWAP(m,v') \Leftrightarrow SORT^{2(n-1)}(M--[m,w])]\ \underline{must}_i$$

then

$$q \rightarrow q' \in R_{2i}[3]$$

□

Proof of Proposition 6.15:

We merely note that the ranking $\langle R_i \rangle_{Nat}$ is such that the statement of Proposition 6.15 with must replaced by R is seen by simple inspection of the pairs [1], [2], and [3] (writing R as a postfix predicate). The result is then immediate by Proposition 6.16.

□

6.3.2.3. Proving part 1

In part 1 of the proof we analyse the possible forms of $t[WS^n(M)]$, for $n > 0$, to discover forms more amenable to manipulation. We begin by noting that we will only have to deal with the definite states of the weavesort machine.

Proposition 6.18

$$t[p]\ \underline{must}\ \text{ iff }\ (\forall\ dt[dp] \in ds(t[p]).\ dt[dp]\ \underline{must})$$

Proof We demonstrate only the reverse implication here; suppose

$$\forall\ dt[dp] \in ds(t[p]).\ dt[dp]\ \underline{must}$$

By Proposition 6.1 and our convention on computations of zero length

$$\langle t_k[p_k] \rangle_K \in comp(t[p])$$

iff

either

$$K = \{0\},\ t_0[p_0] \in ds(t[p]),\ t_0[p_0] \not\rightarrow$$

or

$$K \neq \{0\},\ t[p] = t_0[p_0] \rightarrow t_1[p_1] \rightarrow \ldots$$

In the former case the supposition implies the computation is successful. In the latter case we employ Proposition 6.1 and

Proposition 6.3 to get

$$\nexists\ dt[dp] \in ds(t[p]).\ dt[dp] \rightarrow t_1[p_1] \rightarrow \ldots$$

which must, by the supposition, be a successful computation. The only problem that might arise is if

$$dt[dp] \xrightarrow{\omega!}, \quad t[p] \xrightarrow{\omega!}\!\!\!\!\!\!\not\quad\ $$

but, by Proposition 6.1, this is impossible.

□

This enables us to employ the following proposition when dealing with non-trivial weavesort machines,

Proposition 6.19

Suppose $n > 0$, and $\forall\ t \in T$, $M \in multisets(Nat)$, $|M| \leq 2(n-1)$.

$$t[SORT^{2(n-1)}(M)]\ \underline{must}\ \text{implies}\ t[WS^{n-1}(M)]\ \underline{must}$$

then for any $t \in T$

1. $t[CELLO \Leftrightarrow SORT^{2(n-1)}(\phi)]\ \underline{must}$

   implies

   $t[WS^{n}(\phi)]\ \underline{must}$

2. $t[CELL1(m) \Leftrightarrow SORT^{2(n-1)}(\phi)]\ \underline{must}$

   implies

   $t[WS^{n}([m])]\ \underline{must}$

3. If $M \in multisets(Nat)$, $2 \leq |M| \leq 2n$, $m = min(M)$, then

   $(\forall\ v \in M--[m].\ t[CELL2(m,v) \Leftrightarrow SORT^{2(n-1)}(M--[m,v])]\ \underline{must})$

   implies

   $t[WS^{n}(M)]\ \underline{must}$

Proof

1. Suppose

   $t[CELLO \Leftrightarrow SORT^{2(n-1)}(\phi)]\ \underline{must}$

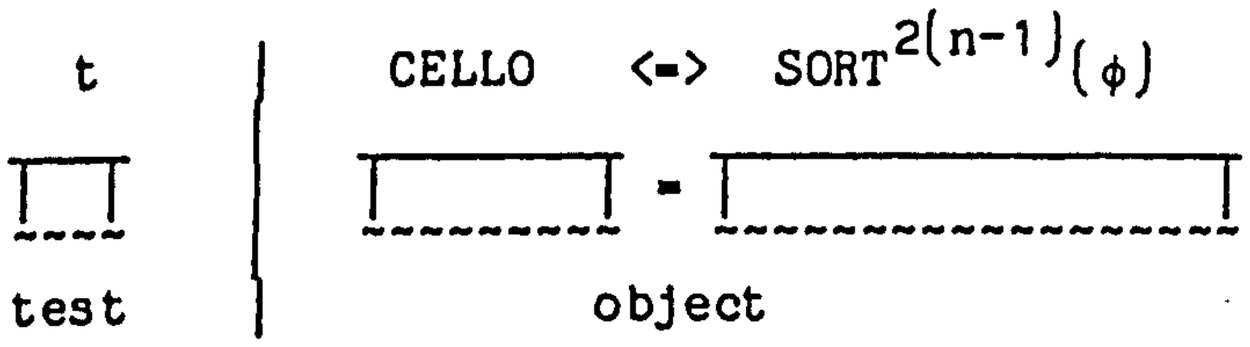
   and take

$$t' = t[CELLO \Leftarrow [.]\ ] \in T$$

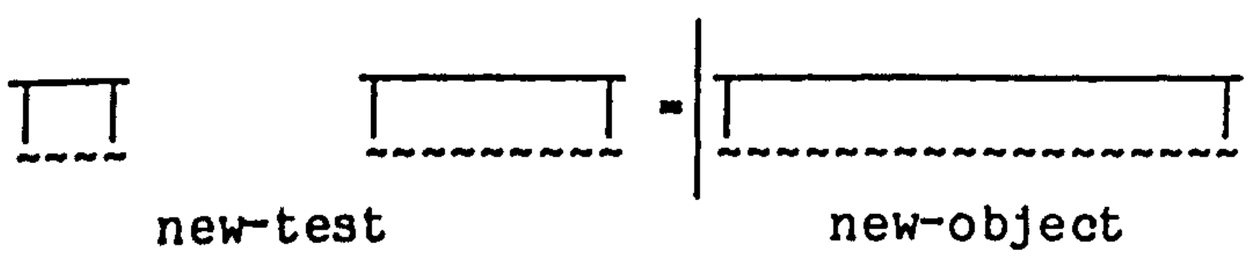
Then

$$t[CELLO \Leftarrow SORT^{2(n-1)}(\phi)] = t'[SORT^{2(n-1)}(\phi)]$$

What we have done is essentially to move the dividing line between test and object, taking part of the object and moving it into the test:



rewrites as



Now, by our supposition we must have

$$t'[SORT^{2(n-1)}(\phi)] \ \underline{must} \ \text{implies} \ t'[WS^{n-1}(\phi)] \ \underline{must}$$

but the antecedent has been shown true, so it just remains to see

$$t'[WS^{n-1}(\phi)] = t[CELLO \Leftarrow WS^{n-1}(\phi)] = t[WS^{n}(\phi)]$$

by moving the dividing line back again.

2. The reasoning is exactly the same as in case 1.

3. Suppose $M \in$ multisets(Nat), $2 \leq |M| \leq 2n$, $m = min(M)$

and

$$\forall\ v \in M--[m].\ t[CELL2(m,v) \Leftarrow SORT^{2(n-1)}(M--[m,v])] \ \underline{must}$$

then

$$\forall\ v \in M--[m].\ t_v[SORT^{2(n-1)}(M--[m,v])] \ \underline{must}$$

where

$$t_v = t[CELL2(m,v) \Leftarrow [.]\ ] \in T$$

so

$\forall$ v ε M--[m]. $t_v[WS^{n-1}(M--[m,v])]$ must

that is

$\forall$ v ε M--[m]. t[CELL2(m,v) <=> $WS^{n-1}(M--[m,v])$] must

Now, applying Proposition 6.18,

$\forall$ v ε M--[m]. $\forall$ dp ε ds($WS^{n-1}(M--[m,v])$). $\forall$ dt ε ds(t).

dt[CELL2(m,v) <=> dp] must

and so by Definition 6.1 and Proposition 6.3

$\forall$ q ε ds($WS^n(M)$). t[q] must

and so by Proposition 6.18 again

$t[WS^n(M)]$ must

□

6.3.2.4. Gathering the parts

We now gather the three parts together as described earlier

Proposition 6.20

Suppose $n \geq 0$, M ε multisets(Nat), $|M| \leq 2n$, then

$\forall$ t ε T. ($t[SORT^{2n}(M)]$ must implies $t[WS^n(M)]$ must)

Proof by induction on n,

BASE n=0: Immediate from the definitions of $WS^0(\phi)$ and $SORT^0(\phi)$

STEP Suppose the result is true for n-1 and

$t[SORT^{2n}(M)]$ must

then by Corollary 6.3

$\exists$ $k \geq 0$. $t[SORT^{2n}(M)]$ $must_k$

We proceed by cases on M,

case $M = \phi$: then by Proposition 6.15

$t[CELL0 <=> SORT^{2(n-1)}(\phi)]$ $must_{2k}$

Now since $must_{2k} \subseteq must$ we can apply the inductive

hypothesis and Proposition 6.19 to get

$t[WS^n(M)]$ must

case $M = [m]$: then by Proposition 6.15

$t[CELL1(m) \iff SORT^{2(n-1)}(\phi)]$ $must_{2k}$

so by the inductive hypothesis and Proposition 6.19

$t[WS^n(M)]$ must

case $2 \leq |M| \leq 2n$, $m = min(M)$: then by Proposition 6.15

$\forall v \in M--[m].\ t[CELL2(m,v) \iff SORT^{2(n-1)}(M--[m,v])]$ $must_{2k}$

so by the inductive hypothesis and Proposition 6.19

$t[WS^n(M)]$ must

□

Corollary 6.4

Suppose $n \geq 0$, $M \in$ multisets(Nat), $|M| \leq 2n$, then

$WS^n(M)$ implements $SORT^{2n}(M)$

□

Conclusion

This thesis has considered two new theories of translation correctness. The contribution of chapters 2 and 3 was the development of a bisimulation theory of translation correctness and a demonstration that it can be applied to non-trivial examples if a sufficiently modular approach is taken. The particular example chosen was an improved version of Li's translation from CSP to CCS, for which we proved result-goodness. The definition of 'result', as the relation of 'goodness', could certainly be strengthened to include notions of convergence and deadlock. Since we already have the bisimulation (which enumerates all the possible states a (CSP or CCS) process can get into) most of the work is already done, it just remains to prove the bisimulation is a subset of the new relation (which presumably is a subset of 'result'). In this way Li's 'correctness' result might be achieved without resort to 'adequacy', but with additional results about the behavioural relationships in the translation.

The main contribution of chapter 4 was the development of the testing equivalence of [de Nicola and Hennessy 84] into a general theory of translation. By contrasting it with a bisimulation theory of translation we were able to attempt a critical comparison of their approaches, wherein a recurrent theme was the problem of trading off nice proof techniques for more interesting translations.

A careful analysis of the testing approach yielded a new notion of correctness called implementation which can be used to capture a relationship of refinement between a specification and the program

that implements it. Later, in chapter 6, we showed through the example of a simple sorting machine in CCS how implementation could be used as a specification tool, and demonstrated a first attempt at a proof technique in CCS.

Much more work needs to be done on implementation before it can be added to the armoury of techniques and concepts commonly used in CCS. We have shown how implementation can define a pre-order over CCS terms but have not given it any axiomatic definition. It seems such a definition should be readily constructable from the work already done on similar pre-orders in [de Nicola and Hennessy 84].

Another direction of research on implementation would be to try giving it a denotational or predicate-transformer interpretation via the work of [Smyth 83] and then comparing it with Park's notion of implementation by determinate 'slices' [Park 81]. This might also enable a comparison, in the denotational style, of the commuting diagram approach to translation correctness [Morris 73] with the testing approach presented here.

Chapter 5 demonstrated some advantages of the testing theory over other theories, most notably its ability to cope with radically different source and target systems. The proof technique employed relied on all computations being finite, though it was shown how this demand could be relaxed.

It is unclear how general this approach is and it would certainly benefit by application to more examples. However, any search for a better proof technique may be seriously hampered by two

problems. Firstly, to cope with very different source and target systems the technique must be stated very abstractly and apply as little constraint as possible; we saw how the constraints of the bisimulation theory prevented its effective application to this example. The second problem lies in the sheer volume of work required to prove even a simple translation, some form of automation is highly desirable if more realistic examples are to be attempted.

In chapter 6 we saw how the CCS operator ⊕ could be used both by the implementer, to (under-) determine a distribution of values over the weavesort machine, and the specifier, to express indifference as to which value is lost upon overflow. As a result of the indeterminacy introduced by ⊕ the relationship established between specification and implementation was _implements_ rather than _completely-implements_ or ≈. Thus the proof had two parts, a _may_ part and a _must_ part.

Proving the _may_ part involved introducing the notion of 'simulation' which has strong similarities to half the bisimulation relator F. It seems quite likely that some weakening of the definition of _simulates_ is desirable , possibly introducing a transition relation ⇒ which is to → as ⇒ is to →. However, it should be noted that the definition of _simulates_ as given exploits the asymmetric relationship of specification and implementation, essentially assuming the specification has no 'silent' internal actions. In many examples this will be so since the specification is often just an abstract requirement of the _external_ behaviour, which might require no reference to 'silent'

action.

The proof of the must part depended (in its inductive step) on being able to rewrite a test on a larger object as a larger test on a (no larger) object. This worked because we could pull a small piece (a single cell) off the (larger) object and into the test. It was very important that the piece was small, for then we could essentially (in the ranking) enumerate its possible states. A thorough investigation is required before this technique can be claimed to have any general applicability.

References

[ADJ 79] J. Thatcher, E. Wagner and J. Wright: More advice on structuring compilers and proving them correct. IBM Research Report RC 7588, 1979.

[Astesiano and Zucca 82] E. Astesiano and E. Zucca: Semantics by translation of CSP and its equivalence with B-semantics: Rapporto Scientifico of the Institute of Math., Genova (Italy), 1982.

[Cardelli 82] L. Cardelli: Real Time Agents. Proc. of the 9'th International Colloquium on Automata, Languages and Programming, Aarhus 1982.

[de Nicola and Hennessy 84] R. de Nicola and M. Hennessy. Testing equivalences for processes. Theoretical Computer Science 34, North Holland 1984.

[Dijkstra 76] E. Dijkstra: A discipline of programming. Prentice Hall, 1976.

[Gordon 79] M. Gordon: The denotational description of programming languages. Springer-Verlag 1979.

[Gordon 81] M. Gordon: Register transfer systems and their behaviour. Proc. of the 5'th International Conference on Hardware Description Languages.

[Guessarian 81] I. Guessarian: Algebraic semantics. in Lecture Notes in Computer Science 99, Springer-Verlag 1981.

[Hennessy and Milner 85] M. Hennessy and R. Milner: Algebraic laws for nondeterminism and concurrency. Journal of the ACM, Vol 32, No 1, 1985

[Hennessy and Plotkin 80] M. Hennessy and G. Plotkin: A term model for CCS. in Lecture Notes for Computer Science 88, Springer-Verlag 1980.

[Hindley et al 72] J. Hindley, B. Lercher and J. Seldin: Introduction to combinatory logic. Cambridge University Press, 1972.

[Hoare 69] C. Hoare: An axiomatic basis for computer programming. Comm. of the ACM, Vol 12, No 10, 1969.

[Hoare 78] C.A.R. Hoare: Communicating Sequential Processes. Communications of the ACM, Vol 21, No 8, 1978.

[Ichbiah et al 79] Rational for the design of the Ada programming language. SIGPLAN Notices 14(b), 1979.

[Jensen and Wirth 78] K. Jensen and N. Wirth: PASCAL user manual and report, Springer-Verlag 1978.

[Landin 63] P. Landin: The mathematical evaluation of expressions. Comp. Journal 6(4), 1963.

[Li 83] W. Li: An operational approach to semantics and translation for concurrent programming languages. PhD Thesis, University of Edinburgh 1983.

[Mead and Conway 80] C. Mead and L. Conway: Introduction to VLSI systems. Addison Wesley, 1980.

[Milne 82] G. Milne: CIRCAL A calculus for circuit description. University of Edinburgh Computer Science report CSR-122-82, 1982.

[Milner 80] R. Milner: A Calculus of Communicating Systems. Lecture Notes in Computer Science 92, Springer-Verlag 1980.

[Milner 82] R. Milner: Calculi for synchrony and asynchrony. Theoretical Computer Science 25, 1983.

[Morris 73] L. Morris: Advice on structuring compilers and proving them correct. Proc. of the ACM conference on Principles of Programming Languages, Boston 1973.

[Mukhopadhyay 81] A. Mukhopadhyay: Weavesort - a new sorting algorithm for VLSI. Technical Report CS-TR-53, Department of Computer Science, University of Central Florida, 1981.

[Park 81] D. Park: A predicate transformer for weak-fair iteration. Proc 6'th IBM Symposium on Math. Foundations of Computer Science, IBM Japan 1981.

[Park 81b] D. Park: Concurrency and automata on infinite sequences. Proc. 5th GI Conference, Lecture Notes in Computer Science, Springer-Verlag 1981.

[Plotkin 81] G. Plotkin: A structural approach to operational semantics. Lecture notes, Aarhus University, 1981.

[Plotkin 82] G.Plotkin: An operational semantics for CSP. Proceedings of IFIP working conference, 1982.

[Stoy 77] J. Stoy: Denotational semantics: the Scott-Strachey approach to programming language theory, MIT Press, 1977.

[Smyth 83] M. Smyth: Powerdomains and predicate transformers: a topological view. University of Edinburgh Computer Science report CSR-126-83, 1983.

Notation

Sets

$\{x: P(x)\}$	intensional definition
$\{x_1, x_2, \ldots\}$	extensional definition
$x \in S$	membership
$x \notin S$	non-membership
$S_1 \subseteq S_2$	subset
$S_1 + S_2$	union
$S_1 - S_2$	difference i.e. $\{x: x \in S_1 \text{ and } x \notin S_2\}$
$S_1 \wedge S_2$	intersection
$\sum_{i \in I} S_i$	indexed union
$S_1 \times S_2$	cartesian product
$P(S)$	powerset
Nat	$\{0,1,2,\ldots\}$

Multisets

$[x: P(x)]$	intensional definition
$[x_1, x_2, \ldots]$	extensional definition
$x \in M$	membership
$x \notin M$	non-membership
$M_1 \subseteq M_2$	submultiset
$M_1 ++ M_2$	multiset union (e.g. $[1,2] ++ [2,3] = [1,2,2,3]$)
$M_1 -- M_2$	multiset difference (e.g. $[1,2,2,2] -- [2,2] = [1,2]$) (Defined only if $M_2 \subseteq M_1$)

Sequences

$\langle c_k \rangle_K$	the sequence $c_{k1}, c_{k2}, \ldots$ where $K = \{k1, k2, \ldots\}$ is a countable set.

$\langle c \rangle_K$ as above.

Relations

rela/tion the relation 'relation' does not hold.

Not interested

Due to the large syntactic size of many terms occuring in this thesis it is useful to substitute '_' for some terms whose exact form is not of immediate interest but is easily deducable from the surrounding context.

Index of Definitions

The format adopted here is to give first the page number at which the definition occurred, second the number of the relevant definition, third the item(s) defined, and lastly any relevant additional information. The entries are organised first by chapters and then by the smallest subsection containing the definition, given in capitals.

6. Weavesort; an example implementation.